Bachelorarbeit

Andre Jestel

Entwicklung einer ModelSim-basierten und verteilten Simulationsumgebung mit SystemC, VHDL und Java

Fakultät Technik und Informatik *Faculty of Engineering and Computer Science*
Department Informatik *Department of Computer Science*

Andre Jestel

Entwicklung einer ModelSim-basierten und verteilten Simulationsumgebung mit SystemC, VHDL und Java

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Michael Schäfers
Zweitgutachter : Prof. Dr. rer. nat. Thomas Canzler

Abgegeben am 23. Juli 2010

Andre Jestel

Thema der Bachelorarbeit

Entwicklung einer ModelSim-basierten und verteilten Simulationsumgebung mit SystemC, VHDL und Java

Stichworte

Interaktive ModelSim-Simluation, Netzwerkzugriff mit SystemC, virtuelle Konsole, Kopplung von Simulationen

Kurzzusammenfassung

Diese Arbeit beschreibt anhand konkret implementierter Anwendungen, wie die VHDL-Simulation mit Hilfe von SystemC verteilt und interaktiv erweitert werden kann. Demonstriert wird, wie die Simulation innerhalb von ModelSim mit einem externen Videospeicher kommuniziert, so dass beliebige Signale während der Simulation visualisierbar werden. Des weiteren wird gezeigt, wie die laufende Simulation durch ein virtuelles Keyboard von außen gesteuert und beeinflusst werden kann. Zusätzlich thematisiert werden Möglichkeiten zur Kopplung von verteilten Simulationen.

Title of the paper

Development of a ModelSim-based and distributed simulation environment in SystemC, VHDL and Java

Keywords

Interactive ModelSim simulation, network access in SystemC, virtual console, interconnection of simulations

Abstract

In this report the distribution and extension of a VHDL simulation by using SystemC on the basis of concrete implemented applications is described. This thesis demonstrates how a simulation in ModelSim communicates with an external Video-RAM so that debug information can be visualized during the simulation. Also the controlling of a running simulation by using an external virtual keyboard is indicated. Additionally possibilities of connecting distributed simulations are discussed.

Inhaltsverzeichnis

Abbildungsverzeichnis....2
Tabellenverzeichnis....3
1 Einleitung....4
1.1 Motivation....4
1.2 Lösungsansatz....5
1.3 Beispielanwendung: Virtuelle Konsole....5
2 SystemC mit ModelSim....7
2.1 Installation....7
2.2 Syntaktisches Gerüst: D-Flip-Flop....8
2.2.1 SystemC-Teil: FlipFlop.cpp....8
2.2.2 VHDL-Teil: TopLevel.vhd....10
2.3 Simulation....12
2.3.1 Tool Command Language....12
2.3.2 Ausführung....13
3 Virtueller Bildschirm....14
3.1 VHDL-Schnittstelle....14
3.1.1 Generische Parameter....14
3.1.2 VHDL-Ports....15
3.2 Adressierung....15
3.3 Transportprotokoll....17
3.4 Kommunikationsprotokoll....18
3.5 Java-Anwendung....20
3.6 Demonstrationstest....21
4 Virtuelles Keyboard....24
4.1 VHDL-Schnittstelle....24
4.2 Steuerung und Timing....25
4.3 Kommunikationsprotokoll....27
4.4 Java-Anwendung....28
4.5 Demonstrationstest....31
5 Kopplung von Simulationen....34
5.1 Taktsynchrone Kopplung....34
5.2 Vollsynchrone Kopplung....35
5.3 Halbsynchrone Kopplung....37
6 Fazit....41
Anhang A – Anwendungsfall: LED-Lauflicht....42
Anhang B – Inhalt der CD....44
Glossar....45
Literaturverzeichnis....46

Abbildungsverzeichnis

Abbildung 1-1: Typisches Schema einer Testumgebung 4
Abbildung 2-1: SystemC-Installation....7
Abbildung 2-2: D-Flip-Flop....7
Abbildung 2-3: SystemC-Quellcode zum D-FlipFlop....8
Abbildung 2-4: Beispielhafte Testumgebung zum D-FlipFlop....10
Abbildung 2-5: TCL-Skript....11
Abbildung 2-6: Simulationsergebnis....12
Abbildung 3-1: VHDL-Schnittstelle des Videospeichers....13
Abbildung 3-2: Initialisierung des Videospeichers....13
Abbildung 3-3: Adressierung des Videospeichers....15
Abbildung 3-4: Sequenzdiagramm vom Lesezugriff....18
Abbildung 3-5: Struktur der VideoRAM-Klasse....19
Abbildung 3-6: Testbench zum Videospeicher....21
Abbildung 4-1: VHDL-Schnittstelle des Keyboards....23
Abbildung 4-2: Initialisierung des Keyboards....23
Abbildung 4-3: Zustandsautomat des Keyboards....25
Abbildung 4-4: ModelSim-Zeit....25
Abbildung 4-5: Sequenzdiagramm einer Zeichen-Abfrage....26
Abbildung 4-6: Struktur der Keyboard-Klasse....27
Abbildung 4-7: Keyboard-Oberfläche vor der Simulation....29
Abbildung 4-8: Testbench zum virtuellen Keyboard....30
Abbildung 4-9: Keyboard-Oberfläche während der Simulation....31
Abbildung 4-10: Keyboard-Oberfläche nach der Simulation....31
Abbildung 4-11: Simulationsergebnis vom virtuellen Keyboard....32
Abbildung 5-1: Beispiel einer synchronen Kopplung....34
Abbildung 5-2: Testbench für bidirektionale Kopplung....36
Abbildung 5-3: Simulationsergebnis mit 8-Byte-Puffergröße....37
Abbildung 5-4: Simulationsergebnis mit 400-Byte-Puffergröße....38
Abbildung 6-1: LED-Lauflicht Schnittstelle....39
Abbildung 6-2: Testbench zum LED-Lauflicht....40
Abbildung 6-3: Lauflicht-Oberfläche nach der Simulation....40

Tabellenverzeichnis

Tabelle 2-1: Auflösungsfunktion der SystemC Logik-Signaltypen....................9
Tabelle 3-1: Ports des Videospeichers....................14
Tabelle 3-2: VHDL-Signalwerte des Videospeichers....................16
Tabelle 3-3: Paketformat für die Kommunikation mit dem Videospeicher....................17
Tabelle 3-4: Kontroll-Byte Aufbau....................17
Tabelle 4-1: Ports des Keyboards....................24
Tabelle 6-1: LED-Beschaltungsvariationen....................39

1 Einleitung

Die bestehenden Möglichkeiten, die ModelSim anbietet, während der Simulation mit anderen Rechnern und Prozessen zu kommunizieren, lassen den Wunsch nach Erweiterungen offen. Aus diesem Bedarf heraus ist diese Bachelorarbeit entstanden. Ziel ist es eine Reihe von Simulationswerkzeugen zu entwickeln, um mit deren Hilfe eine interaktivere Form der Kontrolle über die ModelSim-Simulation zu realisieren. Die dafür verwendeten Methoden ermöglichen verschiedene Verfahren zur Verteilung und Kopplung von Simulationen. Dabei wird durch die Modellierungssprache SystemC [1] eine Schnittstelle erstellt, die es erlaubt während der laufenden Simulation mit ModelSim zu kommunizieren. Über die damit verbundenen Möglichkeiten und Grenzen soll diese Arbeit als ausführlicher Überblick dienen.

Nach der Einleitung in diesem Kapitel folgen in Kapitel 2 die für den SystemC Einsatz notwendigen Vorbereitungen sowie eine exemplarische Erklärung der SystemC Arbeitsweise. Anschließend behandeln Kapitel 3 und 4 ausführlich die Komponenten der virtuellen Konsole. Im Kapitel 5 werden Möglichkeiten bezüglich der Kopplung von verteilten Simulationen erläutert. Sämtliche vorgestellten Projekte sind zusammen mit den Quellcodedateien auf der CD zur Bachelorarbeit zu finden. Eine Übersicht über den Inhalt der CD befindet sich im Anhang.

1.1 Motivation

Die Simulation von Hardwareentwürfen mit Beschreibungssprachen wie VHDL [2] oder Verilog in ModelSim von Mentor Graphics, verläuft im klassischen Sinne deterministisch und unterstützt nur sehr eingeschränkt eine direkte Interaktion mit dem Anwender. In der Regel geschieht die Verifikation der korrekten Funktionsweise eines Entwurfs nach einem Blackbox-Prinzip: An den Eingängen des zu testenden Systems wird eine wohldefinierte Menge von Eingangssignalen angelegt und nach Ablauf der Simulation werden die erzeugten Ausgangssignale mit den Gewünschten abgeglichen. Dieser Ablauf ist nachfolgend illustriert (Abbildung 1-1):

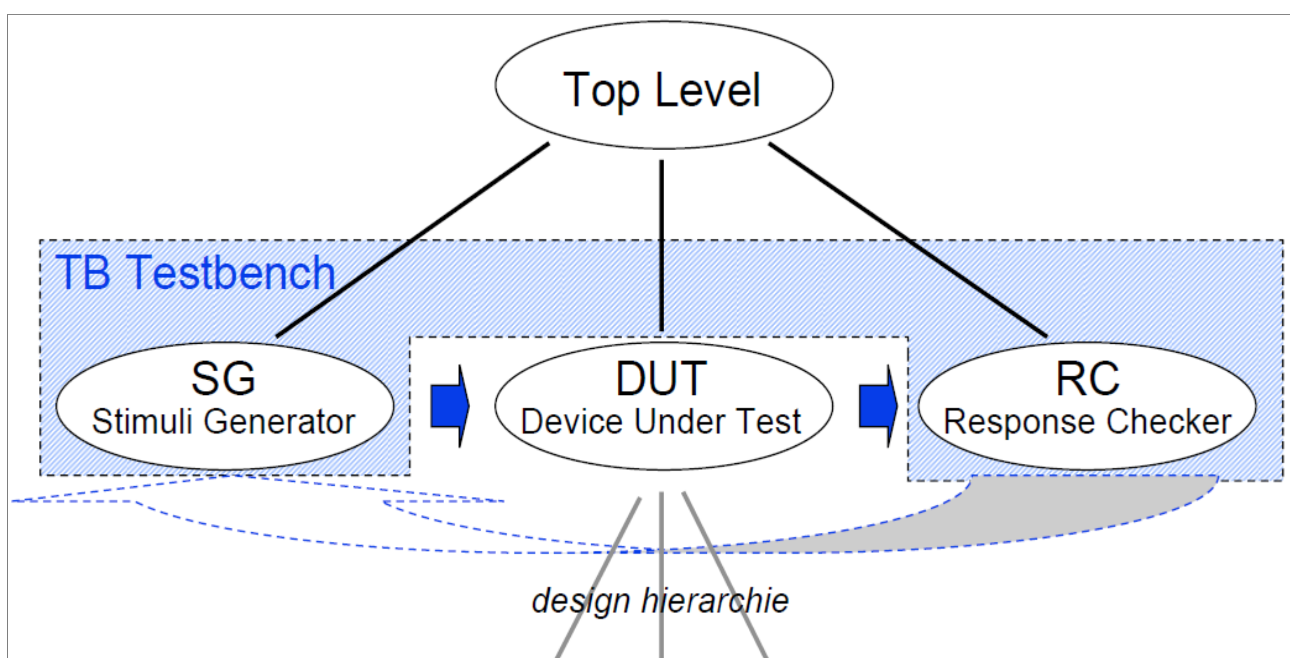


Abbildung 1-1: Typisches Schema einer Testumgebung [14]

Innerhalb einer Testumgebung (Testbench) erzeugt ein Generator die nötigen Eingangssignale, welche an die zu prüfende Einheit (Device Under Test) geleitet werden. Die Ausgangssignale werden anschließend überprüft (Response Checker).

Wünschenswert wäre die Möglichkeit, bereits in der laufenden Simulation, während der Entwickler auf das Ergebnis nach Ablauf der Simulation wartet, komfortableren Einfluss auf das System zu nehmen und Informationen über den momentanen Zustand einsehen zu können.

Darüber hinaus führt die Simulationen von immer komplexer werdenden System, in denen es gilt eine immer größerer Anzahl an nebenläufigen Prozessen zu simulieren, folglich zu stark anwachsenden Anforderungen an die Rechenleistung. Möchte man zum Beispiel eine Menge von Einheiten wie in Mehrkern-Systemen simulieren, wäre es von Vorteil, die Simulationslast von einem Rechner auf mehrere zu verteilen.

Die Recherche nach bereits bestehenden Verfahren, die solche individuellen Erweiterungen oder Kopplungen von ModelSim-Simulation ermöglichen, blieb bisher erfolglos.

1.2 Lösungsansatz

Um eine erweiterte Kontrolle über die ModelSim-Simulation zu ermöglichen, welche die bestehenden Möglichkeiten, die ModelSim mit sich bringt, verbessert, bat es sich an per Socket-API mit der Außenwelt zu kommunizieren. Andere Möglichkeiten zur Kommunikation sind zwar möglich, aber plattformabhängig. Dagegen existiert in allen ModelSim-fähigen Betriebssystemen bereits eine Implementierung des Internet Protocols. Anstelle von betriebssystemabhängigen Mechanismen, bietet die Socket-API eine einfache und transparente Alternative zur Interprozesskommunikation. Daraus ergeben sich aus Entwicklersicht bereits im Vorfeld zwei wesentliche Vorteile:

Ob sich der externe Prozess tatsächlich auf einem anderen physikalischen Rechner befindet oder dieser neben ModelSim auf dem selben Rechner läuft, ist für den Einsatz dieser Kommunikationsschnittstelle unerheblich. Durch das Loopback-Netzwerk im Internet Protocol, welches in IPv4 durch den Adressraum von 127.0.0.1 bis 127.255.255.254 [3] und in IPv6 durch die Adresse ::1 [4] definiert wird, ist die Positionstransparenz gewährleistet. Die einzigen beiden relevanten Verbindungsparameter sind dabei IP-Adresse und Port.

Ein weiterer Vorteil besteht in der Platformunabhängigkeit der Socket-API-Funktionsaufrufen. Die Kommunikationsschnittstelle kann von einer ModelSim-Instanz, welche unter einem Windows-Betriebssystem läuft, genau so genutzt werden wie unter einem Linux-Betriebssystem. Einzige Voraussetzung ist, dass eine ModelSim-Version mit SystemC Unterstützung für das jeweilige Betriebssystem existiert.

1.3 Beispielanwendung: Virtuelle Konsole

Zur anwendungsbezogenen Demonstration der Nutzung dieser Kommunikationsschnittstelle stelle ich eine im Rahmen der Bachelorarbeit entwickelte virtuelle Konsole vor. Diese Konsole besteht aus zwei Teilkomponenten, die unabhängig von einander, alleine oder zusammen eingesetzt werden können.

Die erste Komponente ist die Implementierung eines VGA-typischen 640x480-Monochrom-Display mit Lese- und Schreibzugriff. Der VHDL-Entwickler kann diesen Bildschirmspeicher wie ortsadressierten RAM, also Memory-Mapped, ansprechen. Eine Manipulation des Speichers kann parallel zur Simulation in einem eigenständigen Fenster beobachtet werden.

Bei der zweiten Komponente handelt es sich um ein virtuelles Keyboard, welches ebenfalls in einem eigenständigen Fenster die Eingabe von ASCII-Zeichen ermöglicht, die an die laufende Simulation weitergereicht werden. Dadurch entsteht eine komfortablere Methode auf Benutzereingaben interaktiv in einer eigenständigen Anwendung während der laufenden Simulation zu reagieren.

Beide Anwendungen sind in der Hochsprache Java von Sun Microsystems geschrieben, und sofern für das Zielbetriebssystem eine virtuelle Maschine existiert, können sie plattformunabhängig genutzt werden.

2 SystemC mit ModelSim

Bei SystemC handelt es sich um eine C++ Klassenbibliothek mit dem Ziel Hardwareeigenschaften zu beschreiben. Dazu wurde C++ derart mit syntaktischen Modellierungsmöglichkeiten wie zusätzlichen Makros und Funktionen ausgestattet wurde, dass sie auch zur Simulation von Hardware in der Lage ist. So stehen dem Programmierer neben dem C/C++ Sprachumfang zusätzlich VHDL bzw. Verilog Syntax-Gegenstücke zur Verfügung. Möglich ist es somit auch, die Entwicklung komplexer Systeme über viele Abstraktionsebenen, von Register Transfer Level bis zur grafischen Oberfläche, hinweg ausnahmslos in SystemC durchzuführen. Die IEEE ratifizierte die SystemC Version 2.2 im Jahr 2007 und ist seit dem vollständig mit dem IEEE 1666 - Language Reference Manual konform[5]. Sofern für das Zielbetriebssystem eine entsprechende SystemC-fähige ModelSim-Version verfügbar ist, werden die hier vorgestellten SystemC-Instruktionen gemäß Mentor Graphics, plattformunabhängig unterstützt. Je nach Version und Betriebssystem können die API-Funktionsaufrufe geringfügig variieren.

2.1 Installation

Im Rahmen dieser Arbeit kommt SystemC nicht als Ersatz, sondern als Ergänzung zu VHDL zum Einsatz. Gearbeitet wurde dabei unter Windows XP, sowohl mit ModelSim SE 6.3j als auch mit Version 6.5d. Um den Funktionsumfang von SystemC mit ModelSim unter Windows nutzen zu können, bedarf es den MinGW-gcc-Compiler[6], der den C/C++ Quellcode übersetzt und als Bibliothek mit dem Simulations-Kernel verlinkt. Für ModelSim 6.3j ist gcc-Version 3.3.1 und für 6.5d die gcc-Version 4.2.1 erforderlich.

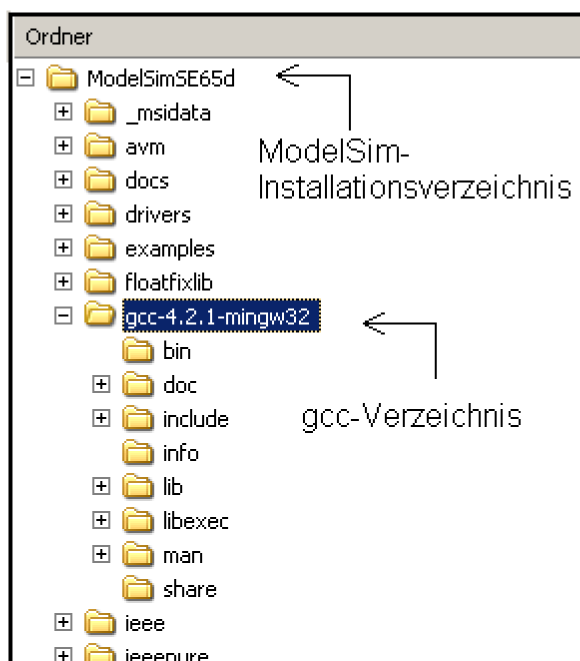


Abbildung 2-1: SystemC-Installation

Zur Installation genügt es das entsprechende gcc-Verzeichnis in das ModelSim-Wurzelverzeichnis zu kopieren (siehe Abbildung 2-1). Dabei ist der syntaktisch exakte Verzeichnisname zu beachten: „gcc-4.2.1-mingw32“ bei ModelSim-Version 6.5d bzw. „gcc-3.3.1-mingw32“ bei Version 6.3j. Für die hier genutzten Funktionen konnten keine Unterschiede zwischen den Versionen festgestellt werden. Sowohl der Videospeicher als auch das virtuelle Keyboard wurden mit beiden Versionen erfolgreich getestet und sind uneingeschränkt nutzbar.

2.2 Syntaktisches Gerüst: D-Flip-Flop

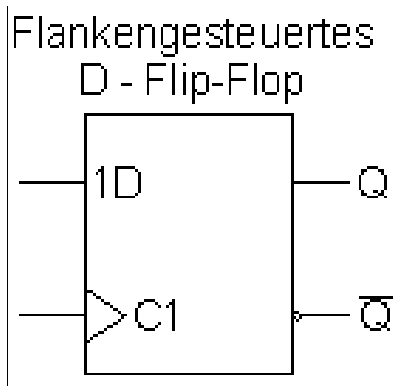


Abbildung 2-2: D-Flip-Flop

Zur Demonstration des Zusammenspiels von VHDL und SystemC soll ein einfaches D-FlipFlop (Abbildung 2-2) dienen. Es kann als ein syntaktischer Minimalrahmen für den Einsatz von SystemC betrachtet werden, welcher bereits einen Großteil der Syntax-Elemente enthält, die bei den späteren Anwendungen Verwendung finden.

2.2.1 SystemC-Teil: FlipFlop.cpp

Als erster Schritt soll das Erstellen der benötigten SystemC-Komponente vorgestellt werden (Abbildung 2-3):

```
1  #include <systemc.h>
2
3  SC_MODULE( FlipFlop ){
4      sc_in< sc_logic > clk, reset;
5      sc_in< sc_logic > d;
6      sc_out< sc_logic > q;
7
8      void prc_flipflop() {
9          if ( reset == '1' ) {
10             q = sc_dt::Log_0;
11         } else if ( clk.posedge() ) {
12             q = d;
13             cout << "Steigende Takt-Flanke @ " << sc_time_stamp() << endl;
14         }
15     }
16
17     SC_CTOR( FlipFlop ) {
18         SC_METHOD( prc_flipflop );
19         sensitive << clk.pos() << reset;
20     }
21 };
22
23  SC_MODULE_EXPORT(FlipFlop)
```

Abbildung 2-3: SystemC-Quellcode zum D-FlipFlop

Dieser Aufbau unterscheidet sich grundsätzlich vom reinen SystemC mit eigenen Simulations-Kernel. Der Schwerpunkt liegt hier deutlich auf der Kooperation und Koexistenz mit ModelSim und VHDL.

Zeile 1 enthält eine Präprozessor Include-Anweisung, welche dem gcc-Compiler vor dem eigentlichen Übersetzungsvorgang, die nötigen SystemC-Makros, Typdeklarationen und Funktionen bekannt gibt. Zeile 3 entspricht dem VHDL-Entity/Architecture Gegenstück. Das **SC_MODULE**-Makro deklariert eine C++-Klasse mit dem in Klammern angegebenen Namen. Dabei ist das Semikolon hinter der schließenden Klammer in Zeile 21 notwendig und zu beachten. Die Zeilen 4 bis 6 stellen die Port-Deklaration dar, mit denen das Flip-Flop mit den VHDL-Komponenten kommunizieren kann.

Bei dem Inhalt der eckigen Klammern handelt es sich um den Datentyp, für den dieser Port verwendet werden kann. SystemC kennt die nativen C/C++ Datentypen: **`(unsigned) int`**, **`(unsigned) long int`**, **`(unsigned) short int`**, **`double`**, **`float`**, **`(unsigned) char`**, und **`bool`**. Zusätzlich zur Verfügung stehen unter anderem spezielle Datentypen für ganze Zahlen beliebiger Größen wie **`sc_bigint<>`** und **`sc_biguint<>`**, sowie für Gleitkommazahlen: **`sc_fix<>`** und **`sc_ufix<>`**. Die eckigen Klammern enthalten dabei die Anzahl an Bits, die den Wertebereich definieren. Kommen bei der Auswahl der Datentypen mehrere Varianten in Frage, empfiehlt es sich aus Performance-Gründen die nativen C/C++ Datentypen zu nutzen [1].

Als Gegenstück zu den VHDL-Standard-Logik-Signaltypen finden überwiegend **`sc_logic`** und **`sc_lv<>`** für Logik-Vektoren Verwendung, wo bei die Bitbreite der Vektoren in den Klammern angegeben wird. Sie bestehen aus 4-wertigen Bits mit folgender Auflösungsfunktion (Tabelle 2-1), die bei mehreren Signaltreibern zur Anwendung kommt:

Bedeutung	Wert	'X'	'0'	'1'	'Z'
Undefiniert	'X'	'X'	'X'	'X'	'X'
Logische Null	'0'	'X'	'0'	'X'	'0'
Logische Eins	'1'	'X'	'X'	'1'	'1'
Hochohmig	'Z'	'X'	'0'	'1'	'Z'

Tabelle 2-1: Auflösungsfunktion der SystemC Logik-Signaltypen

In den Zeilen 8 bis 15 befindet sich eine Methodendefinition der zuvor deklarierten Flip-Flop-Klasse. Wie innerhalb von VHDL-Prozessen, werden die Anweisungen sequentiell abgearbeitet. In diesem Fall wird das Verhalten, eines auf die positive Taktflanke reagierenden D-Flip-Flops mit Reset-Eingang, in C/C++ - Syntax beschrieben. In der optionalen Ausgabe in Zeile 13 wird zusätzlich die SystemC-Funktion **`sc_time_stamp()`** aufgerufen, die als Rückgabewert den aktuellen Simulationszeitpunkt mit Einheit als Text-String zurück liefert.

Das **SC_CTOR()-**Makro, welches sich von Zeile 17 bis 20 erstreckt, definiert den Konstruktor der Klasse. Dort werden dem Scheduler alle für die Simulation relevanten Informationen bekannt

gegeben. Das **SC_METHOD()**-Makro teilt dem Simulator mit, dass er die in den Klammern angegebene Funktion wie einen VHDL-Prozess mit Empfindlichkeitsliste behandelt. Direkt darauf folgen durch **`sensitive`**-Anweisungen welche Signale den jeweiligen Prozess anstoßen. Der **<<** - Operator wurde dazu aus Gründen der Lesbarkeit überschrieben[7]. Alternativ ließe sich das gleiche Verhalten durch eine **`sensitive(SignalName)`**- Anweisung erreichen. Allerdings ist jeweils nur genau ein Signal pro Anweisung möglich, während sich bei der ersten Variante mehrere Signale übersichtlich konkatenieren lassen. Soll, wie in diesem Fall bei der Taktleitung, nicht auf den Signalpegel, sondern auf steigende oder fallende Flanken reagiert werden, ist dem entsprechenden Signal noch **.pos()** bzw. **.neg()** anzufügen. Die konkrete Flanken-Abfrage erfolgt wie in Zeile 11 durch Aufruf der **.posedge()** bzw. **.negedge()** Methoden der Signale. Sie liefern boolsche Wahrheitswerte je nachdem, ob sich im aktuellen Delta-Zyklus eine entsprechende Flanke ereignet hat.

Das in der letzten Zeile verwendete Makro kennzeichnet Top-Level Design-Units, die somit nach ModelSim exportiert werden [8]. Es ordnet die SystemC-Einheiten in der Hierarchie den VHDL-Einheiten unter. Bei ausschließlicher Nutzung von SystemC ohne ModelSim und VHDL wäre dieses Makro nicht notwendig. Weiterhin sorgt es auch dafür, dass eine entsprechend passende VHDL-Entity automatisch vom SystemC-Compiler erstellt wird, die von ModelSim für das Zusammenspiel mit VHDL benötigt wird. Möchte man bei der Entwicklung die Deklarationen in Header-Dateien (.h) von den Definitionen trennen, ist das **SC_MODULE_EXPORT() -**Makro nur an das Ende der Header-Datei zu schreiben.

2.2.2 VHDL-Teil: TopLevel.vhd

Der nächste Schritt zeigt wie, das in SystemC entworfene D-Flip-Flop, als abgeschlossene Komponente in die VHDL-Design-Hierarchie integriert wird (Abbildung 2-4):

```
 1 LIBRARY ieee;
 2 USE ieee.std_logic_1164.ALL;
 3
 4 LIBRARY work;
 5 USE work.ALL;
 6
 7 ENTITY TopLevel IS
 8 END ENTITY;
 9
10 ARCHITECTURE behavior OF TopLevel IS
11
12 SIGNAL clk_s, reset_s, d_s, q_s : std_logic;
13
14 COMPONENT FlipFlop IS
15    PORT (
16       clk, reset, d : IN std_logic;
17       q : OUT std_logic);
18 END COMPONENT FlipFlop;
19
20 BEGIN
21    my_FlipFlop : FlipFlop PORT MAP ( clk_s, reset_s, d_s, q_s );
22
23    test_process : PROCESS
24    BEGIN
25       reset_s <= '1';
26       clk_s <= '0';
27       d_s <= '1';
28       WAIT FOR 10 ns;
29
30       reset_s <= '0';
31       WAIT FOR 10 ns;
32
33       clk_s <= '1';
34       WAIT;
35    END PROCESS;
36
37 END ARCHITECTURE behavior;
```

Abbildung 2-4: Beispielhafte Testumgebung zum D-FlipFlop

Die Zeilen 1 bis 5 ermöglichen den Zugriff auf die Standard-Logik-Signaltypen, sowie auf die eigenen Design-Einheiten zu denen auch das D-Flip-Flop selbst gehört. Da es sich hier um eine Testumgebung handelt, tauchen in der Entity-Deklaration (Zeile 7 und 8) keine weiteren Ports mehr auf. In Zeile 12 werden jene lokale Signale deklariert, die den Eingangs- und Ausgangssignalen des Flip-Flops entsprechen. Die **COMPONENT-**Anweisung in Zeile 14 bis 18 stellt die Schnittstelle zum Flip-Flop dar. Es tauchen die zuvor in SystemC geschriebenen Signale mit ihrem Namen, Datentyp und ihrer Richtung auf. Nach der **Architecture-BEGIN-**Anweisung folgt in Zeile 21 per **PORT MAP** das Abbilden der lokalen Signale auf die Ein- und Ausgänge des Flip-Flops.

Abschließend folgt in einem VHDL-Prozess das Generieren der Signalezustände:

- Zeitpunkt 0 ns: Initialisieren der Anfangssignale mit gleichzeitigen Reset des Flop-Flops
- Zeitpunkt 10 ns: Freigabe der Reset-Leitung
- Zeitpunkt 20 ns: Erzeugen einer steigenden Taktflanke

Das letzte **WAIT-**Statement ohne Zeitangabe dient dazu, die Simulation später durch den ModelSim-Befehl: „run -all" von Anfang bis zum Ende laufen zu lassen. Ansonsten würden die Anweisungen im Prozess unendlich oft wiederholt werden.

2.3 Simulation

Nachdem der Quellcode angefertigt wurde, folgt nun das Kompilieren und Ausführen der Simulation. Zuerst ist in ModelSim ein reguläres VHDL-Projekt anzulegen. Anschließend sind dem Projekt die angefertigten Quellcode-Dateien: „FlipFlop.cpp" und „TopLevel.vhd" hinzuzufügen. Die zur Simulation nötigen Schritte können durch zahlreiche Buttons und Menü-Dialoge erledigt werden. Allerdings ist diese manuelle Befehlsausführung, insbesondere bei Verwendung von SystemC-Komponenten, sehr umständlich und daher für Entwicklungs- und Programmierzyklen nicht zu empfehlen.

2.3.1 Tool Command Language

Deutlich mehr Komfort bietet die Nutzung der *Tool Command Language*, kurz **TCL**. Ursprüngliches Ziel dieser Sprache war es, eine universelle, plattformunabhängige Skriptsprache für alle bestehenden Entwicklungswerkzeuge zu erschaffen. Sie ermöglicht die automatische Abarbeitung der ModelSim-Befehle und bietet gegenüber der manuellen Ausführung eine beträchtliche Zeiteinsparung. In das Wurzelverzeichnis des angelegten VHDL-Projektes ist dazu eine .tcl – Datei zu Erstellen. Die benötigten Befehle, sowie ihre Reihenfolge (Abbildung 2-5) sind identisch mit jenen Befehlen, die zur Nutzung der virtuellen Konsole notwendig sind.

```
1   vlib work
2
3   vdel -allsystemc
4
5   sccom *.cpp
6
7   vcom *.vhd
8
9   sccom -link
10
11  vsim TopLevel
```

Abbildung 2-5: TCL-Skript

Der **vlib** - Befehl zu Anfang stellt sicher, dass die „work"-Designbibliothek existiert. Fehlt sie, wird das zugehörige Verzeichnis zusammen mit zugehörigen Index-Dateien erstellt. Ist die Existenz dieser Bibliothek bereits im Vorfeld sicher gestellt, kann der Befehl weggelassen werden. Werden im Laufe der Programmierung SystemC-Komponenten geändert, gelöscht oder hinzugefügt, sollte dafür gesorgt werden, dass stets nur die aktuellen Komponenten kompiliert und zur Simulation genutzt werden. Da das manuelle Entfernen veralteter SystemC-Designeinheiten nicht möglich ist, wird dies automatisch durch **vdel -allsystemc** erledigt.

Das eigentliche Kompilieren einzelner oder mehrerer Quellcode-Dateien erfolgt dann bei SystemC-Dateien durch **sccom** und bei VHDL-Dateien durch **vcom**. Aufgrund der Tatsache, dass es sich bei der obersten Hierarchieebene um einen VHDL-Entwurf handelt, in den der SystemC-Entwurf eingegliedert werden soll, müssen die SystemC-Dateien zuerst vor dem VHDL-Quellcode kompiliert werden.

Damit Quellcode-Änderungen zur Simulation übernommen werden ist es notwendig, dass der

SystemC-Linker stets eine aktuelle systemc.so - Objektdatei anlegt, die während der Simulation geladen wird. Für die Erweiterung durch externe Funktionen müssen an dieser Stelle, nach der -link – Option, die zusätzlichen C/C++ - Bibliotheken hinzugefügt werden[9]. Abschließend wird der ModelSim-Simulator mit dem zu simulierenden Top-Level-Entwurf durch vsim <Designname> gestartet.

2.3.2 Ausführung

Das TCL-Skript im Projekt-Verzeichnis kann im ModelSim-Transcript-Fenster durch **source <Skriptdatei>** ausgeführt werden. Danach ist die Simulation startbereit und kann per **run** – Befehl angestoßen werden. Zuvor können die relevanten Signale ins Wave-Fenster hinzugefügt werden. Damit die Simulation bis zum Ende abläuft kann dem **run** – Befehl noch die **-all** – Option angefügt werden. Anhand des Simulationsergebnisses (Abbildung 2-6) lässt sich die korrekte Arbeitsweise des D-Flip-Flops verifizieren:

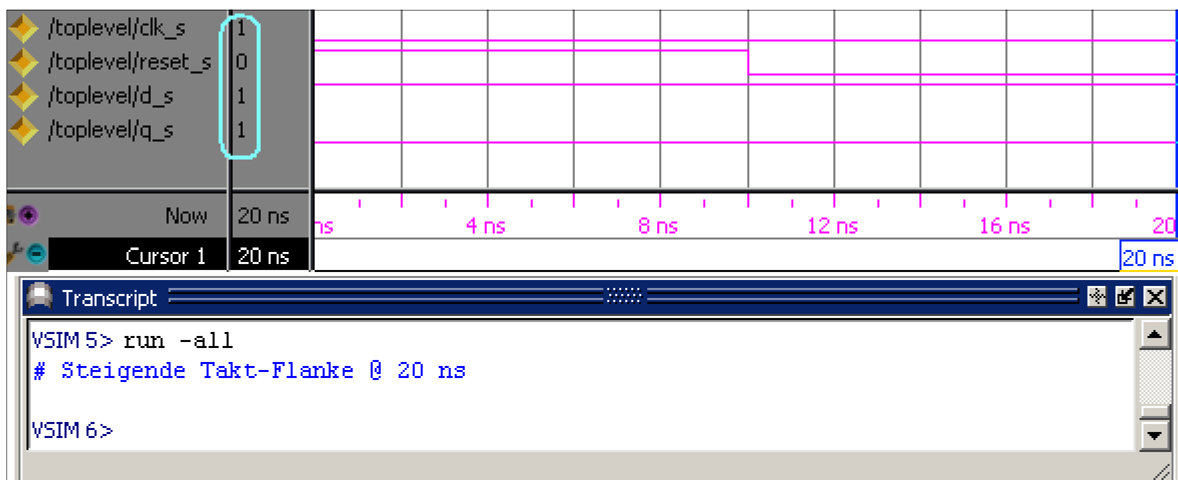


Abbildung 2-6: Simulationsergebnis

Zum Zeitpunkt t = 20 ns wurde die anliegende logische '1' durch die steigende Taktflanke vom Flip-Flop übernommen. Gleichzeitig kann durch die Textausgabe das Ausführen des C/C++ - Quellcodes beobachtet werden.

3 Virtueller Bildschirm

Ziel dieser Anwendung ist es, sich aus Sicht des VHDL-Entwicklers wie ein Hardware-Videospeicher nutzen zu lassen. Es entsteht der Eindruck, als handle es sich um einen tatsächlich physikalischen Speicher, dessen Inhalt als virtueller Bildschirm einzusehen ist. Hinter der VHDL-Schnittstelle sorgt SystemC mit der Socket-API für die Kommunikation mit der Java Software-Komponente des Videospeichers. Die Speicherzugriffe sind parallel zur Simulation in der Java-Anwendung beobachtbar.

3.1 VHDL-Schnittstelle

Zur Ansteuerung des Videospeichers über VHDL steht die folgende Schnittstelle (Abbildung 3-1) zur Verfügung:

```
COMPONENT vRAM_interface -- IP-Address, UDP-Port and Baseaddress
   GENERIC ( ip_addr1, ip_addr2, ip_addr3, ip_addr4, udp_port,
             base_addr_high, base_addr_low : integer );
   PORT (
      nCS_p     : IN std_logic; -- Chip Select (active low)
      RDxnWR_p  : IN std_logic; -- Read-Write Access
      vaccess_p : IN std_logic_vector(3 DOWNTO 0);     -- Access Select
      vaddr_p   : IN std_logic_vector(31 DOWNTO 0);    -- Address
      vdata_p   : INOUT std_logic_vector(31 DOWNTO 0); -- Data
      nReset_p  : IN std_logic); -- Reset (active low)
END COMPONENT vRAM_interface;
```

Abbildung 3-1: VHDL-Schnittstelle des Videospeichers

Um die Adressierung des Speichers möglichst flexibel zu halten, dient das *vaccess_p* Signal zum einzelnen Auswählen der vier möglichen Bytes bei einem Zugriff.

3.1.1 Generische Parameter

Die hinter dem GENERIC-Schlüsselwort [2] befindlichen Parameter sind zur Konfiguration einmalig beim Initialisieren notwendig. Damit werden IP-Adresse, Port und Anfangsadresse des Speichers festgelegt. Befindet sich die Java-Anwendung zum Videospeicher auf dem selben Rechner, auf dem auch die ModelSim-Simulation läuft und soll der Adressraum beispielhaft bei 0xFFC00000 beginnen, könnte das Einbinden des Speichers wie folgt aussehen (Abbildung 3-2):

```
my_vRAM : vRAM_interface
GENERIC MAP (127, 0, 0, 1, 10500, 16#FFC0#, 0 )
PORT MAP ( nCS_s, RDxnWR_s, vaccess_s, vaddr_s, vdata_s, nReset_s );
```

Abbildung 3-2: Initialisierung des Videospeichers

Die ersten vier Werte stellen die IP-Adresse des Rechners, auf die Videospeicher Java-Anwendung läuft, in *dotted decimal notation* dar. Der fünfte Wert ist der Port auf dem die Java-Anwendung die

Datenpakete empfängt. Dabei muss sichergestellt sein, dass dieser gewählte Port nicht bereits von anderen Anwendungen belegt ist. Die beiden letzten Parameter repräsentieren die Anfangsadresse des Speichers. Aufgrund des Wertebereichs muss die Adresse in zwei Teile aufgespalten werden. Im Beispiel soll die Startadresse auf 0xFFC00000 gelegt werden. Der höherwertige Teil 0xFFC0 wird als vorletzter Parameter angegeben. Zum Schluss noch der niederwertige Anteil (hier 0) entsprechend.

3.1.2 VHDL-Ports

Nachfolgend (Tabelle 3-1) sind sämtliche Signale des Speichers aufgeführt:

Signalname	Bit-Breite	Beschreibung
nCS_p	1	Chip Select (active low): Muss auf logisch '0' liegen damit Lese- und Schreibzugriffe ausgeführt werden können. Eine logische '1' kann zur Vorbereitung eines Zugriffs genutzt werden.
RDxnWR_p	1	Lese- oder Schreibzugriffs Auswahl: '1' – Lesezugriff '0' – Schreibzugriff (Datenleitung wird hochohmig geschaltet)
vaccess_p	4	Byteauswahl: Jedes der vier Bits steuert ein Datenbyte. '1' – Byte wird geschrieben oder gelesen '0' – Byte wird nicht geschrieben oder gelesen
vaddr_p	32	Adressleitung: Anfangsadresse + Offset zur Adressierung des Speichers.
vdata_p	32	Datenleitung: Beim Lesezugriff wird der Inhalt des Speichers auf diese Leitung gelegt. Dabei müssen die anderen Busteilnehmer die Datenleitung hochohmig schalten. Beim Schreibzugriff werden hier die zu schreibenden Daten angelegt.
nReset_p	1	Reset-Leitung (active low): '0' - Löschen aller Bits im Speicher. '1' - Normalbetrieb

Tabelle 3-1: Ports des Videospeichers

3.2 Adressierung

Der Videospeicher verfügt mit dem Bildschirm über 640x480 Schwarz-Weiß Pixel, wobei jedes Pixel durch ein Bit repräsentiert wird. Nach einem Reset sind alle Bits gelöscht und der Bildschirm ist somit komplett schwarz. Der Lese- und Schreibzugriff erfolgt über einen 32 Bit Datenbus Byteweise in variabler Breite von 8 Bit bis 32 Bit. Es gilt stets die *Little Endian* Byte-Reihenfolge [10] so dass das niederwertigste Bit des niederwertigsten Bytes an der kleinsten Adresse steht.

Die Anfangsadresse kann beim Einbinden des Speichers in den VHDL-Quellcode beliebig gewählt werden, so dass der Speicher über einen 32 Bit Adressbus als zusätzliche Komponente betrieben

werden kann, ohne das es zu Adress-Konflikten kommt. Auf diese Basisadresse wird zur Auswahl der Pixel ein Adressen-Offset dazu addiert. Der Adressraum wird dann durch folgende Offsets auf dem Bildschirm abgebildet (Abbildung 3-3):

0x0000	0x0001	0x0002	...	0x004E	0x004F
0x0050	0x0051	...	...	...	0x009F
0x00A0	...	...	...	...	...
...	...	...	...	...	...
0x95B0	...	...	...	...	0x95FF

Abbildung 3-3: Adressierung des Videospeichers

Es sind insgesamt $\frac{640 \times 480}{8} = 38.400 = 0x9600$ Bytes adressierbar.

Damit die Zugriffsmethode möglichst flexibel bleibt und kompatibel zu gängigen Zugriffsarten, insbesondere der des ARM-Mikroprozessors [11], erfolgt die Auswahl der Bytes durch *One-Hot-Kodierung* [2]. Jedes Bytes des Datenbusses, wird durch genau ein Bit ausgewählt oder ignoriert. Sind alle Bits gesetzt sind 32 Bit Zugriffe möglich und bei 2 gesetzten Bits 16 Bit Zugriffe. Insgesamt sind somit $2^4 = 16$ mögliche Zugriffsmuster pro Lese- oder Schreibvorgang möglich und es existiert keine *alignment*-Beschränkung Dadurch sind mehr Varianten möglich als die der ARM unterstützt und bleibt es dem Anwender freigestellt, welche Zugriffsarten er für seine Zwecke nutzt.

Je nach Zugriffsbreite ergeben sich dadurch unterschiedliche Varianten um z.B. auf ein und das selbe **zusammenhängende** Wort zuzugreifen:

- 32 Bit → genau eine Variante
- 24 Bit → zwei Varianten: Basisadresse + Offset und 0b0111 = 0x7 als Byteauswahl oder Basisadresse + Offset – 1 und 0b1110 = 0xE als Byteauswahl
- 16 Bit → drei Varianten: 0b0011, 0b0110 oder 0b1100 mit jeweils um 1 verschobenen Adressen
- 8 Bit → vier Varianten: Jeweils genau ein Bit in der Auswahl gesetzt

Zusätzlich sind **zusammenhangslose** Worte durch entsprechende Auswahl möglich, wie z.B. nur das kleinst- und höchstwertige Byte.

Beispielhaft folgen die VHDL-Signalwerte, die zum Setzen der vier Eckpixel nötig bzw. möglich sind (Tabelle 3-2). Der Zugriff erfolgt byteweise und als Startadresse wurde 0xFFC00000 gewählt:

	Links oben	Rechts oben	Links unten	Rechts unten
nCS_p	'0'	'0'	'0'	'0'
nReset_p	'1'	'1'	'1'	'1'
vaccess_p	"0001"	"0001"	"0001"	"0001"
RDxnWR_p	'0'	'0'	'0'	'0'
vdata_p	x"00000001"	x"00000080"	x"00000001"	x"00000080"
vaddr_p	x"FFC00000"	x"FFC0004F"	x"FFC095B0"	x"FFC095FF"

Tabelle 3-2: VHDL-Signalwerte des Videospeichers

3.3 Transportprotokoll

Die Videospeicher-Zugriffe werden von SystemC auf einzelne Datenpakete umgesetzt. Zu Beginn der Simulation wird in einen einmalig angestoßenen SC_THREAD ein *User Datagram Protocol* (kurz UDP) - Endpunkt (Socket) erstellt [12]. Im Gegensatz zu dem verbindungsorientierten *Transmission Control Protocol (* kurz TCP) fiel die Entwurfsentscheidung für UDP als Transport-Protokoll aus, da die verbindungsorientierten Eigenschaften nicht erforderlich und auch zum Teil nicht erwünscht sind. Zusammengefasst ergeben sich dadurch folgende Vor- und Nachteile:

+ Erhöhte Skalierbarkeit
+ Geschwindigkeitsgewinn
- Paketverlust bleibt unbemerkt

TCP erfordert einen expliziten Verbindungsaufbau und -abbau. Fällt die Java-Anwendung selbst oder die Netzwerkverbindung zum virtuellen Videospeicher während einer ggf. mehrstündigen Simulation kurzzeitig aus, so bedarf es keines kompletten Neustarts der Verbindung. Die Java-Anwendung kann permanent auch zwischen Simulationen weiterlaufen und auch von mehreren, verteilten Simulationen genutzt werden. Dadurch wird eine bessere Skalierbarkeit erreicht.

Durch Wegfall der Fluss- und Staukontrollmechanismen, die TCP u.a. mit dem 3-Wege-Handshake und einen größeren Header gegenüber UDP bietet, können einzelne Pakete durch weniger Protokoll-Overhead schneller transportiert werden. Dies kann sich insbesondere bei vielen Zugriffen in der Geschwindigkeit bemerkbar machen.

Sicherheitsmechanismen sind bei UDP im Vergleich zu TCP zwar deutlich reduziert, aber bei dieser Anwendung nicht kritisch. Sollte ein fehlerhaftes Paket empfangen werden, ist dies sehr wahrscheinlich in Form eines falsch gesetzten Pixel zuerkennen. Möglich bleibt es auch weiterhin das UDP-Prüfsummenfeld beim Empfang zu nutzen ohne das Transportprotokoll wechseln zu müssen. Weiterhin wächst die Wahrscheinlichkeit, dass einzelne Pakete verloren gehen, proportional zur Auslastung und Ausdehnung des Netzes. Da der Einsatz des Videospeichers für überschaubare lokale Netze ausgelegt ist, sind Paketverluste sehr unwahrscheinlich.

3.4 Kommunikationsprotokoll

Damit das Zusammenstellen, der Transport und die Auswertung der Datenpakete möglichst schnell abläuft, gilt es sie so kompakt und einfach wie möglich zu halten. Hierzu habe ich das Kommunikation in 9-Byte langen Paketen realisiert (Tabelle 3-2):

Byte-Position	9. bis 6.	5. bis 2.	1.
Beschreibung	32 Bit Daten	32 Bit Speicheradresse	Kontrollbyte

Tabelle 3-3: Paketformat für die Kommunikation mit dem Videospeicher

Die kleine Größe dieses Paketformats garantiert, dass jeder Transport weit unterhalb der *Maximum Transmission Unit* (kurz, MTU) bleibt und somit keinen Geschwindigkeitsverlust aufgrund von IP-Fragmentierung erleidet. Weiterhin wird zur Verkürzung der Bearbeitungszeit auf ein Prüfsummenfeld verzichtet.

Für die Byte-Reihenfolge der Daten- und Adress-Bytes gilt weiterhin *Little Endian,* so dass das niederwertigste Daten- bzw. Adress-Byte an der jeweils kleinsten Position im Sende- und Empfangsbuffer steht. Das Kontroll-Byte an der allerersten Position dient zur En- und Decodierung der Zugriffsart (Tabelle 3-3):

Bit-Position	8. bis 6.	5. bis 2.	1.
Beschreibung	unbenutzt (mit Ausnahme beim Reset)	Byte-Auswahl: Jedes Bit steht für ein Daten-Byte. Ist es gesetzt wird wird das entsprechende Daten-Byte gelesen bzw. geschrieben. Bei der Zuordnung gilt *Little Endian*.	Gesetzt: Lesezugriff Gelöscht: Schreibzugriff

Tabelle 3-4: Kontroll-Byte Aufbau

Einen Spezialfall stellt das Übertragen eines Resets, im Falle eines Low-Pegels des nReset_p-Signals, dar. Dann sind alle acht Bits im Kontroll-Byte gesetzt, so dass ein Reset in einer logischen Abfrage durch Vergleich mit '-1' festgestellt werden kann. Die restlichen Bytes für Daten und Speicheradresse werden dann ignoriert.

Die Zusammenstellung der Bytes erfolgt in dem SystemC-Prozess, der flankensensitiv auf die Signale: **nCS_p, RDxnWR_p** und **nReset_p** reagiert. Die anderen Signale müssen zum Zeitpunkt der fallenden Chip-Select-Flanke korrekt anliegen. Abwechselnde Lese- und Schreibzugriffe durch Flankenwechsel des **RDxnWR_p** Signals ohne zwischenzeitliches Aufheben des Chip-Select-Pegels, sind durch die Delta-Zyklen von VHDL auch möglich. Der Sendepuffer wird anschließend nach der Aktivierung unter Verwendung der C/C++ Schiebeoperationen << und >> mit den Daten gefüllt.

Im Falle eines Resets oder Schreibzugriffs erfolgen die Sendevorgänge asynchron zur Simulation, so dass diese nach dem Abschicken der Pakete sofort weiterläuft. Bei einem Lesezugriff blockiert die Simulation im aktuellen Delta-Zyklus solange bis die Antwort mit den entsprechenden Daten von der Java-Anwendung eingetroffen ist (Abbildung 3-4):

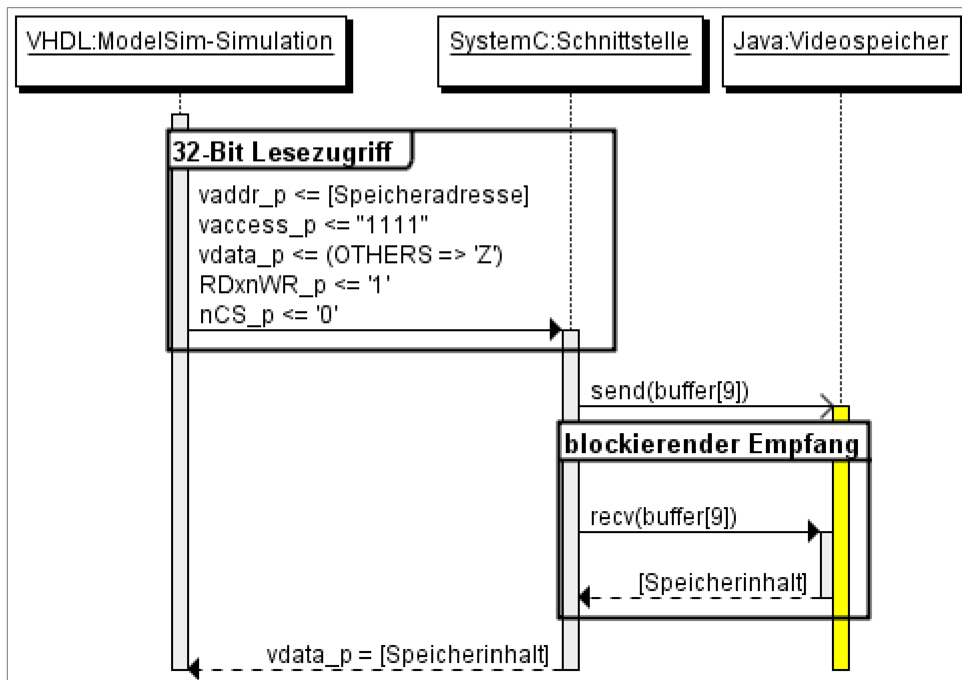


Abbildung 3-4: Sequenzdiagramm vom Lesezugriff

Unabhängig von der tatsächlichen physikalischen Laufzeit, die das Paket und die Antwort brauchen, passiert ein Lesezugriff aus Sicht der Simulation **unmittelbar**.

3.5 Java-Anwendung

Die Visualisierung des Videospeichers, in der sich auch der Speicher selbst befindet, wurde von mir im Rahmen dieser Arbeit in Java 6 Update 18 geschrieben und getestet. Sämtliche Java-API Klassen und Methoden, auf die zugegriffen, wird standen aber bereits seit der Java Version 1.0 zur Verfügung, was Abwärtskompatibilität zu älteren Laufzeitumgebungen garantiert. Auf die Verwendung von bestehenden GUI-Buildern oder Grafik-Pattern wurde aufgrund der überschaubaren Anforderungen verzichtet.

Die Anwendung selbst besteht aus einer Klasse **VideoRAM**, die dem folgenden Aufbau entspricht (Abbildung 3-5):

```
package virtualConsole;

import java.awt.*;
import java.awt.event.*;
import java.net.*;

public class VideoRAM extends Frame implements Runnable {

    static final int WIDTH = 640, HEIGHT = 480;
    byte[] vRAM = new byte[WIDTH*HEIGHT/8];

    DatagramSocket udpSocket;

    VideoRAM(int udp_portnum ) {

    @Override
    public void paint(Graphics g) {

    @Override
    public void run() {

    public static void main(String[] args){
}
```

Abbildung 3-5: Struktur der VideoRAM-Klasse

Als globale Attribute verfügt die Klasse über den Speicher selbst in Form eines Byte-Arrays und über ein **Datagram-Socket -** Objekt, welches für den Empfangs- und Sendevorgang genutzt wird. Im Konstruktor **VideoRAM()** wird das Socket – Objekt mit der übergebenen Portnummer initialisiert und Attribute, die die Klasse von der **Frame**-Klasse geerbt hat, wie Fenstergröße oder Hintergrundfarbe werden dort festgelegt.

Die von der **Frame**-Klasse überschriebene Methode **paint()**, wird bei jedem Zeichnen des Fensters ausgeführt und sorgt dafür, dass alle im Videospeicher gesetzten Bits in Form von weißen Pixel auf dem Bildschirm erscheinen.

Durch die Implementierung des **Runnable**-Interface und überschreiben der **run()**-Methode kann

beim Erzeugen des Objektes ein eigenständiger Thread mit erzeugt werden. Dieser wartet empfangsblockierend in einer Endlosschleife und kümmert sich um das Empfangen, Auswerten und ggf. Beantworten von UDP-Paketen, die von der SystemC-Schnittstelle der Simulation abgeschickt werden. Ein Erben von der Thread-Klasse selbst ist an dieser Stelle nicht möglich, da Java keine reine Mehrfachvererbung unterstützt.

Wie beim Zusammenstellen der Pakete geschieht auch die Analyse der Pakete durch Schiebeoperationen und Bit-Verknüpfungen. Eine Terminierung der Anwendung ist nur durch das explizite Schließen durch den Anwender vorgesehen. So kann die Software wie ihr Hardware-Gegenstück zwischen den Simulationen weiter laufen und ohne zusätzlichen Reinitialisierungs-Aufwand weiter genutzt werden. Des Weiteren führt auch ein ein Zugriffsfehler auf den Speicher durch ein **Try-Catch** – Statement nicht zum Absturz der Anwendung. Mögliche Fehler werden abgefangen, eine entsprechende Fehlermeldung wird in der zur Java-Anwendung gehörenden Konsole ausgegeben und die Endlosschleife wird fortgesetzt.

Als Einstiegspunkt der Anwendung dient die **main()**-Methode. Sie erzeugt aus der Klasse das aktive Objekt und übergibt als Kommandozeilenparameter den UDP-Port, auf dem der Empfangs-Thread auf eingehende Pakete von der Simulation wartet. Wird kein Parameter beim Start mit übergeben, so wird als Defaultwert die Portnummer 10500 gewählt. Vom Anwender muss sichergestellt werden, dass der ausgewählte UDP-Port nicht von einem anderen Prozess benutzt wird. Zur Laufzeit wird der genutzte Port im oberen breiten Fensterrahmen angezeigt.. Der Aufruf über die Kommandozeile oder mit Hilfe einer Batch-Datei lautet entsprechend:

java virtualConsole.VideoRAM <UDP-Portnummer>

Bei „virtualConsole“ handelt es sich um den Paketnamen und gleichzeitig Dateiordner, in dem sich die „VideoRAM.class“ und „VideoRAM$1.class“ Dateien befinden.

Die Anwendung muss vor dem Start der ModelSim-Simulation bereits laufen.

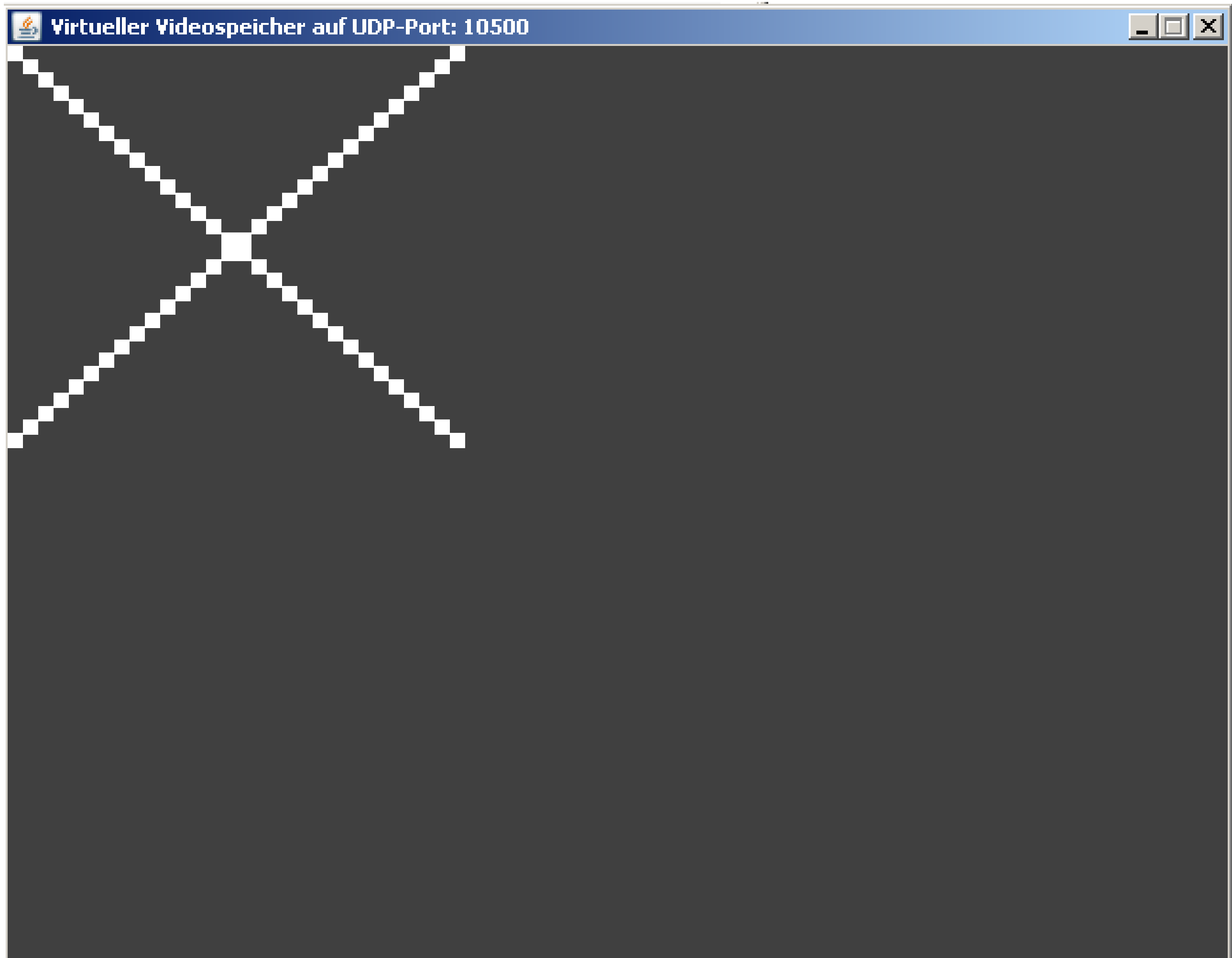
3.6 Demonstrationstest

Die korrekte Funktionsweise des virtuellen Videospeichers soll mit Hilfe einer Testumgebung (Abbildung 3-6) überprüft werden. Angefangen in der links-oberen Ecke bei der Adresse 0xFFC00000 werden nacheinander 60 Quadrate, bestehend aus je 8*8 gesetzten Pixel, in den Videospeicher geschrieben. Dabei werden die Adressen so gewählt, dass zwei diagonale Linien entstehen, die zusammen ein Kreuz in der links-oberen Ecke bilden. Ein einzelnes Quadrat wird durch 8 aufeinanderfolgende Byte-Schreibzugriffe in einer For-Schleife realisiert. In jedem Durchlauf wird die Adresse für die erste Diagonale um 80 erhöht, so dass sich ein Quadrat von Oben nach Unten aufbaut. Entsprechend umgekehrt verhält es sich bei der zweiten Diagonalen, die von links-unten anfängt.

```
my_vRAM : vRAM_interface
GENERIC MAP (127, 0, 0, 1, 10500, 16#FFC0#, 0 )
PORT MAP ( nCS_s, RDxnWR_s, vaccess_s, vaddr_s, vdata_s, nReset_s );

 test_process : PROCESS
     VARIABLE i,j : integer;      -- Zählervariablen
 BEGIN
     nCS_s <= '1';                -- Vorbereiten
     RDxnWR_s <= '0';             -- Schreibzugriff
     vaccess_s <= "0001";         -- Ein Byte pro Zugriff
     vdata_s <= x"000000FF";      -- Immer 8 Bits setzen
     vaddr_s <= x"FFC00000";      -- Oben-Links anfangen
     nReset_s <= '0';             -- Speicher löschen
     WAIT FOR 10 ns;
     nReset_s <= '1';             -- Reset aufheben

     FOR i IN 1 TO 30 LOOP        -- Länge je 30 Quadrate
         FOR j IN 1 TO 8 LOOP     -- 8 mal 8 Pixel pro Quadrat
             nCS_s <= '0';        -- Schreiben
             WAIT FOR 10 ns;
             vaddr_s <= vaddr_s + 80;
             nCS_s <= '1';        -- Schreiben vorbereiten
             WAIT FOR 10 ns;
         END LOOP;
         vaddr_s <= vaddr_s - 80 + 1;
     END LOOP;

     vaddr_s <= x"FFC041A0";      -- 2. Gerade vorbereiten
     FOR i IN 1 TO 30 LOOP        -- Länge je 30 Quadrate
         FOR j IN 1 TO 8 LOOP     -- 8 mal 8 Pixel pro Quadrat
             nCS_s <= '0';        -- Schreiben
             WAIT FOR 10 ns;
             vaddr_s <= vaddr_s - 80;
             nCS_s <= '1';        -- Schreiben vorbereiten
             WAIT FOR 10 ns;
         END LOOP;
         vaddr_s <= vaddr_s + 80 + 1;
     END LOOP;
     WAIT;
 END PROCESS;
```

Abbildung 3-6: Testbench zum Videospeicher

Nach dem erfolgreichen Durchlauf, sieht die Oberfläche des Videospeichers wie folgt aus (Abbildung 3-7):



4 Virtuelles Keyboard

Während der virtuelle Bildschirm dem Anwender Speichermanipulationen visualisiert, dient das virtuelle Keyboard dazu, ASCII-Zeichen von der Tastatur an die laufende Simulation zu übermitteln. Die Oberfläche der zugehörigen Java-Anwendung stellt die bereits verarbeiteten sowie die noch zu sendenden Zeichen dar. Wie beim Videospeicher sorgt eine SystemC-Schnittstelle für die Kommunikation mit der Java-Anwendung. Dabei kommt ebenfalls UDP als Transport-Protokoll zum Einsatz.

4.1 VHDL-Schnittstelle

Die Steuerung des Keyboards erfolgt durch folgende Schnittstelle (Abbildung 4-1):

```
COMPONENT vKeyboard_interface IS -- IP-Address and UDP-Port
   GENERIC ( ip_addr1, ip_addr2, ip_addr3, ip_addr4, udp_port : integer );
   PORT (
      kdata_p     : OUT std_logic_vector(7 DOWNTO 0); -- ASCII-Code
      kvalid_p    : OUT std_logic; -- Valid Character
      kconsu_p    : IN std_logic;  -- Character Consumed
      krequest_p  : IN std_logic;  -- Request Character
      nReset_p    : IN std_logic); -- Reset (active low)
END COMPONENT vKeyboard_interface;
```

Abbildung 4-1: VHDL-Schnittstelle des Keyboards

IP-Adresse und der UDP-Port zur Java-Anwendung werden über die generischen Parameter wieder in *dotted decimal notation* angegeben. Das Zuweisen der Ports auf lokale Signale und das Initialisieren der Keyboard-Schnittstelle könnten wie folgt (Abbildung 4-2) aussehen:

```
my_vKeyboard : vKeyboard_interface
GENERIC MAP (127, 0, 0, 1, 10600 )
PORT MAP (kdata_s, kvalid_s, kconsu_s, krequest_s, nReset_s );
```

Abbildung 4-2: Initialisierung des Keyboards

Analog zum Videospeicher muss der Anwender sicherstellen, dass der ausgewählte UDP-Port nicht bereits von einer anderen Anwendung belegt ist, insbesondere wenn der Videospeicher zusammen mit dem Keyboard genutzt werden soll.

Die Schnittstelle verfügt über zwei Ausgangs- und drei Eingangssignale, deren Bedeutungen in der nachfolgenden Tabelle 4-1 aufgelistet wird:

Signalname	Bit-Breite	Richtung	Beschreibung
kdata_p	8	OUT	Aktueller ASCII-Code: Bis zum Empfang des ersten Zeichens, sind alle Bits gelöscht. Danach bleibt das Zeichen bis zum nächsten Empfang des Nächsten angelegt.
kvalid_p	1	OUT	Gültigkeits-Bit: '1' – Der anliegende ASCII-Code ist gültig und wurde noch nicht verarbeitet. '0' – Der anliegende ASCII-Code ist nicht gültig oder wurde bereits verarbeitet.
kconsu_p	1	IN	Verarbeitungs-Bit: Eine steigende Flanke signalisiert dem Keyboard, dass das aktuelle ASCII-Zeichen verarbeitet wurde.
krequest_p	1	IN	Anfrage-Bit: Eine steigende Flanke signalisiert demm Keyboard eine Nachfrage, ob ein neues Zeichen im Puffer vorliegt.
nReset_p	1	IN	Reset-Leitung (active low): '0' - Löschen aller Zeichen im Keyboard-Puffer '1' - Normalbetrieb

Tabelle 4-1: Ports des Keyboards

4.2 Steuerung und Timing

Zur Kontrolle des Zeichenempfangs sind im Wesentlichen die beiden flankensensitiven Eingangssignale **kconsu_p** und **krequest_p** verantwortlich. Durch diese wird der SystemC-Prozess der Keyboard-Schnittstelle und ein Zustandsautomat (Abbildung 4-3) mit zwei Zuständen gesteuert. Der Wert des Gültigkeits-Bits entspricht dabei jeweils genau einen Zustand. So führt die steigende Flanke des Verarbeitungs-Bits nur dann zu einem Zustandswechsel, wenn zu diesem Zeitpunkt ein gültiges Zeichen anliegt. Umgekehrt wird die steigende Flanke des Anfrage-Bits nur dann verarbeitet, wenn kein gültiges Zeichen (mehr) anliegt. In diesem Moment findet ein Dialog mit der korrespondierenden Java-Anwendung statt. Je nachdem, ob während des Dialogs mindestens ein neues Zeichen vorhanden ist oder nicht, wird der entsprechende Folgezustand eingenommen. Unabhängig davon führt ein Reset immer in den „Zeichen ungültig“- Zustand.

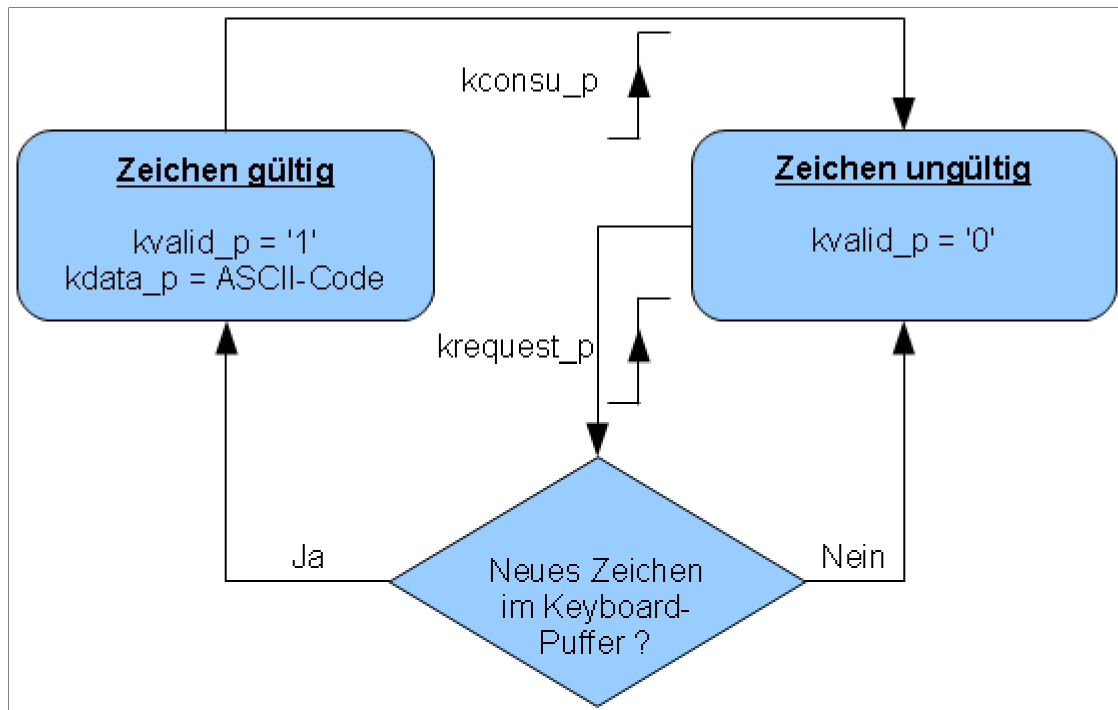


Abbildung 4-3: Zustandsautomat des Keyboards

Es wird an dieser Stelle deutlich, dass der Empfang von Zeichen auf Anfrage (*Polling*) erfolgt. Zunächst könnte eine ereignisorientierte Methode, welche die Zeichen unmittelbar nach der Eingabe auf der Tastatur, ohne Anfrage, an die Simulation weiterleitet, als geeigneter erscheinen. Aufgrund des Zeitverhaltens des ModelSim-Schedulers, ist dies nicht praktikabel umsetzbar. Denn der Simulator verfügt über einen internen Zähler, welcher den aktuellen Zeitpunkt der Simulation in der ModelSim-Simulationsansicht angibt (Abbildung 4-4).

Abbildung 4-4: ModelSim-Zeit

Wie viel physikalische Zeit während dieser *virtuellen* Zeit tatsächlich vergeht, ist ausschließlich von der Komplexität des zu simulierenden Systems, sowie der Rechnerleistung abhängig. Wenn es in keinem VHDL-Prozess mehr zu Signaländerungen kommt, endet die Simulation. Da ModelSim über keinerlei Informationen verfügen kann, ob irgendwann (noch) eine Tastatureingabe erfolgt, muss der Empfang zu diskreten Zeitpunkten erfolgen. Ansonsten könnte der interne Zähler überlaufen oder aber der Zeitpunkt des Empfangs schwankt stets durch die Faktoren Rechnerleistung und Simulationskomplexität. Der Empfang von Zeichen zu einen gewünschten Zeitpunkt, der zur korrekten Simulation erforderlich ist, wäre aufgrund dieser Unvereinbarkeit von physikalischer und *virtueller* Zeit nicht gewährleistet. Dieser Umstand ist im Wesentlichen für die Einschränkungen bei gekoppelten Simulationen verantwortlich und wird auch im 5. Kapitel diskutiert.

Je nach Anwendungsfeld muss der Benutzer daher für einen entsprechenden Anfrage-Takt über das **krequest_p** Signal sorgen. Wird dieser Takt zu hochfrequent gewählt für dies zu einer längeren

Simulationszeit, da jede Anfrage den internen Zähler bis zum Empfang der Antwort anhält. Eine zu niederfrequente Auswahl führt zu einem unkontrollierteren Empfangszeitpunkt.

Durch diese synchrone Abfrage kann aber ein ereignisorientierter Empfang (Interrupt) von Tastatureingaben aus Simulations-Sicht realisiert werden. Wird eine synchrone Hardware wie beispielsweise ein µProzessor simuliert, so kann der Abfrage-Takt an die Prozessor-Taktleitung gekoppelt werden. Dadurch ist sichergestellt, dass der Prozessor über das **kvalid_p** Signal sofort auf ein eingegebenes Zeichen reagieren kann.

Liegt nun ein Zeichen im Tastaturpuffer vor, wird dieses, wie bei einem Lesezugriff auf den Videospeicher, bei der Abfrage ohne Verzögerung an die Simulation übermittelt. Somit ist es möglich, unmittelbar vor dem Start der Simulation, die gewünschten Zeichen einzugeben, so dass diese nacheinander abgeholt werden können. Liegt dagegen kein Zeichen vor, wird die laufende Simulation für eine auswählbare Zeit angehalten. Erfolgt dann eine Eingabe innerhalb dieses Zeitfensters, wird es sofort übermittelt und die Simulation wird fortgesetzt. Bleibt die Eingabe aus, wird die Simulation nach Ablauf des Zeitfensters mit „kvalid_p = 0“ fortgesetzt.

4.3 Kommunikationsprotokoll

Analog zu den zeitlichen Anforderungen des Videospeichers, gilt es hier umso mehr, dass die Verarbeitung der UDP-Pakete so schnell wie möglich erfolgt. Zur Realisierung der Kommunikation reicht ein einziges Byte. Sind alle Bits in diesem Byte gelöscht, handelt es sich um eine Anfrage nach neuen Zeichen. Sind dagegen alle Bits gesetzt, wird der Java-Anwendung ein Reset signalisiert. Die ebenfalls aus einem Byte bestehende Antwort enthält entweder den ASCII-Code des vorliegenden Zeichens oder aber den Wert 0x00, welcher das Nichtvorhandensein eines neuen Zeichens repräsentiert. Der zeitliche Ablauf einer erfolgreichen Anfrage wird im folgenden Sequenzdiagramm (Abbildung 4-5) visualisiert:

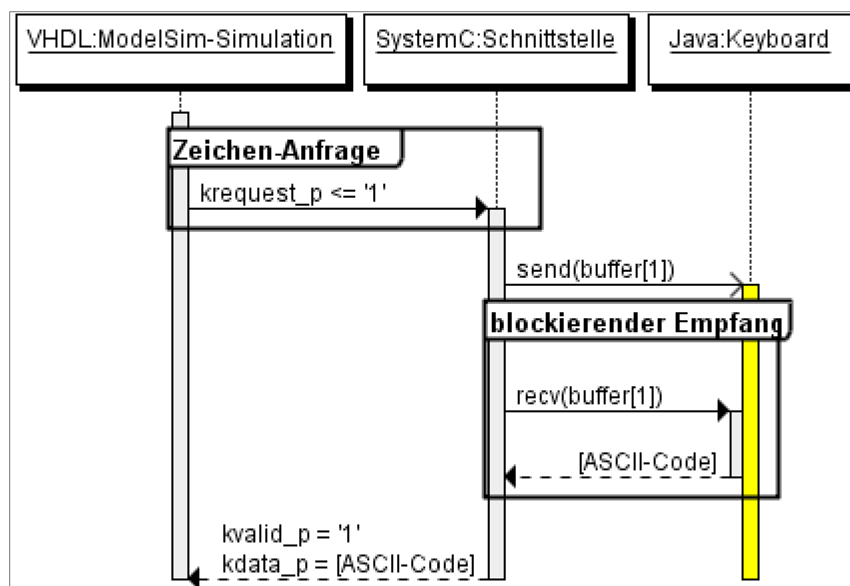


Abbildung 4-5: Sequenzdiagramm einer Zeichen-Abfrage

4.4 Java-Anwendung

Zur Aufnahme der eingegebenen Zeichen dient eine Java-Fenster-Anwendung. Entwickelt und getestet wurde wieder mit der Java-Version 6 Update 18. Parallel zum Empfang der Pakete muss die Anwendung ebenfalls zu jedem Zeitpunkt auf Tastatureingaben reagieren können. Zu diesem Zweck nutzt sie eine spezielle Datenstruktur zur Synchronisation, die erst seit der Java-Version 5 zum Bestandteil der Java-API gehört[13]. Somit ist die Abwärtskompatibilität bis zu dieser Version garantiert.

Anhand des Aufbaus der **Keyboard**-Klasse (Abbildung 4-6), aus der die Anwendung besteht, folgt die Erklärung der Arbeitsweise.

```
package virtualConsole;

import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.LinkedList;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Keyboard extends Frame implements Runnable, KeyListener {

    DatagramSocket udpSocket;

    ConcurrentLinkedQueue<Byte> charQueue
    LinkedList<Byte> sendedQueue

    Thread recvThread;
    int timeout;
    boolean isRequesting;

    Keyboard(int udp_portnum, int timeout ) {

    @Override
    public void paint(Graphics g) {

    @Override
    public void run() {

    @Override
    public void keyPressed(KeyEvent e) {

    public static void main(String[] args) {
}
```

Abbildung 4-6: Struktur der Keyboard-Klasse

Zur Darstellung der Fenster-Oberfläche erbt die Klasse von der **Frame**-Klasse, deren **paint()**-Methode, die bei jedem Neuzeichnen des Fenster aufgerufen wird, zur Visualisierung der Zeichenpuffer überschrieben wurde. Der Empfang, die Auswertung und das das Beantworten der UDP-Pakete erfolgt in einem eigenständigen Thread, der in der überschriebenen **run()**-Methode aus dem **Runnable**-Interface implementiert ist.

Eingegebene Zeichen werden in der überschriebenen **keyPressed()**-Methode aus dem **KeyListener**-Interface, in Form von **KeyEvent**-Objekten, als Parameter empfangen und als Byte in die **ConcurrentLinkedQueue** geschoben. Diese FIFO-Datenstruktur zum Verwalten von Zeichen übernimmt die Synchronisation bei parallelen Zugriffen von mehreren Threads. Bei alternativer Verwendung von Datenstrukturen, die nicht 'thread-safe' sind, wären eine Verletzung der Reihenfolge bzw. Kausalität als auch Laufzeitfehler möglich. Kritisch wird das nebenläufige Ausführen der Operationen auf die Queue insbesondere dann, wenn ein Eingabeereignis parallel zum Empfang eines Zeichenanfrage-Paketes eintritt. Eine ebenfalls sichere Variante würde die *BlockingQueue* aus dem **java.util.concurrent**-Package darstellen. Allerdings ist ihre Kapazität statisch festgelegt und im Gegensatz zur verwendeten Queue blockierend, wenn eine Operation die Kapazität über- oder unterschreiten würde. Daraus folgt eine erhöhte Gefahr für ungewolltes Blockieren der Anwendung und eine Einschränkung des Zeichenpuffers. Durch die Fähigkeit der LinkedQueue bei Bedarf dynamisch zu wachsen, ist die Möglichkeit eine beliebige Anzahl von Zeichen zwischen-zuspeichern nur durch die Speicherverwaltung der *Java Virtual Machine* begrenzt.

Ist die Queue zum Zeitpunkt einer Zeichenanfrage gerade (noch) leer, legt sich der Empfangs-Thread für eine definierte Zeit (**timeout**) schlafen. Eine Tastatureingabe, die innerhalb dieses Zeitfenster stattfindet, soll den Kommunikationsdialog mit ModelSim sofort wieder fortsetzen können. Hierzu dient die globale Speicherung des **recvThread**-Objekts, welches über den Aufruf der interrupt()-Methode den schlafenden Thread wieder vorzeitig aufwecken kann.

Bis eine Anfrage nach einem neuen Zeichen erfüllt wurde, wird dies dem Anwendung durch einen roten Schriftzug: „Zeichen-Nachfrage“ signalisiert. Dieser Zustand wird der **paint()**-Methode durch die boolesche **isRequesting**-Variable übermittelt.

Jedes Zeichen, dass an die ModelSim-Simulation übermittelt wurde, wird aus der **charQueue** entnommen und in die **sendedQueue** umgepackt. Diese Queue enthält stets die letzten 13 gesendeten Zeichen und werden dem Benutzer angezeigt.

Bei der Darstellung der Zeichen wird stets der ASCII-Code, und bei druckbaren Zeichen auch das Zeichen selber, dargestellt (Abbildung 4-7):

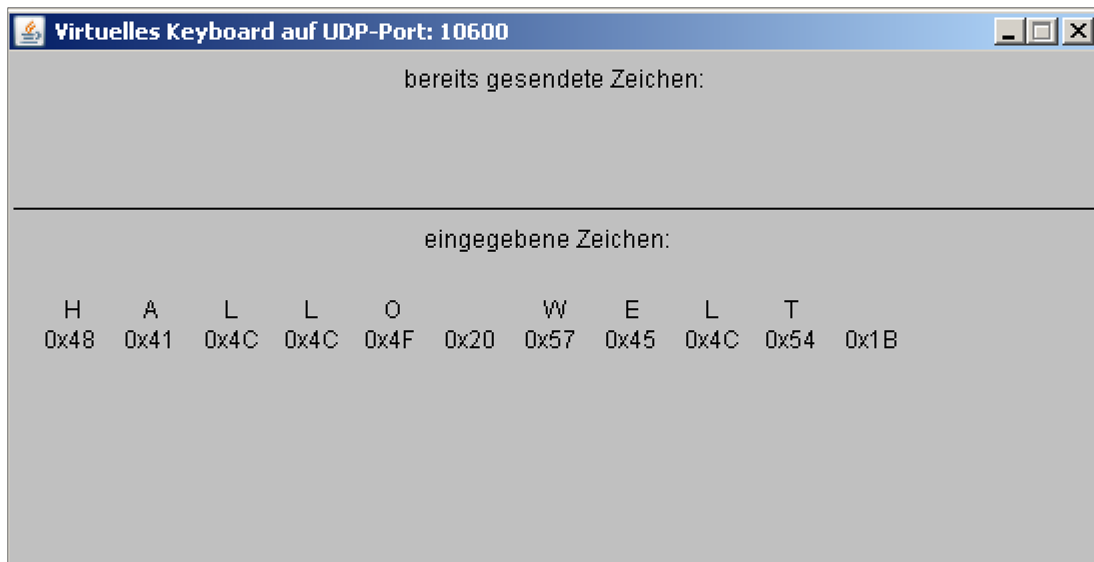


Abbildung 4-7: Keyboard-Oberfläche vor der Simulation

Als Einstiegspunkt der Anwendung dient die **main()**-Methode. Sie erzeugt aus der Klasse das aktive Objekt und übergibt als Kommandozeilenparameter den UDP-Port und die Wartezeit in Millisekunden, die auf maximal auf Eingaben gewartet wird. Werden keine Parameter beim Start mit übergeben, so wird als Defaultwert die Portnummer 10600 und 300ms als Wartezeit gewählt. Vom Anwender muss sichergestellt werden, dass der ausgewählte UDP-Port nicht von einem anderen Prozess benutzt wird. Zur Laufzeit wird der genutzte Port im oberen breiten Fensterrahmen angezeigt. Der Aufruf über die Kommandozeile oder mit Hilfe einer Batch-Datei lautet entsprechend:

java virtualConsole.Keyboard <UDP-Portnummer> <Wartezeit>

Bei„virtualConsole“ handelt es sich um den Paketnamen und gleichzeitig Dateiordner, in dem sich die „Keyboard.class“ und „Keyboard$1.class“ Dateien befinden.

Die Anwendung muss vor dem Start der ModelSim-Simulation bereits laufen.

4.5 Demonstrationstest

Als einfache Überprüfung der korrekten Arbeitsweise, soll die folgende Testumgebung dienen (Abbildung 4-8). Ein VHDL-Prozess fragt solange zyklisch nach neuen Zeichen und konsumiert diese, bis der Anwender die Escape-Taste (ASCII-Code: 0x1B) gedrückt hat. Dabei wird der Kommunikationsdialog zwischen VHDL und der Java-Anwendung verdeutlicht.

```
my_vKeyboard : vKeyboard_interface
GENERIC MAP (127, 0, 0, 1, 10600 ) -- IP, Port
PORT MAP (kdata_s, kvalid_s, kconsu_s, krequest_s, nReset_s );

test_process : PROCESS
BEGIN
   -- Initialiserungen
   kconsu_s <= '0';
   krequest_s <= '0';
   nReset_s <= '0';
   WAIT FOR 10 ns;
   nReset_s <= '1';

   -- Hauptschleife
   WHILE (kdata_s /= x"1B") LOOP -- Solange bis Escape empfangen
     krequest_s <= '1'; -- Zeichen-Nachfrage
     WAIT FOR 10 ns;
     krequest_s <= '0'; -- Steigende Flanke vorbereiten

     IF ( kvalid_s = '1' ) THEN -- Liegt gültiges Zeichen an ?
       kconsu_s <= '1'; -- Zeichen 'konsumieren'
     END IF;
     WAIT FOR 10 ns;

     kconsu_s <= '0'; -- Steigende Flanke vorbereiten
   END LOOP;

   krequest_s <= '1'; -- Letzte Anfrage
   WAIT; -- Simulationsende
END PROCESS;
```

Abbildung 4-8: Testbench zum virtuellen Keyboard

Damit ein erneutes Nachfragen und verbrauchen der Zeichen möglich ist, müssen die Signale wieder zurück gesetzt werden, da steigende Flanken erforderlich sind. Nach dem Start der Simulation wurde die Zeichenkette: „HALLO WELT“ mit abschließenden Escape-Zeichen eingegeben. Der Ablauf der Sendevorgänge ist auf der grafischen Oberfläche beobachtbar (Abbildung 4-9).

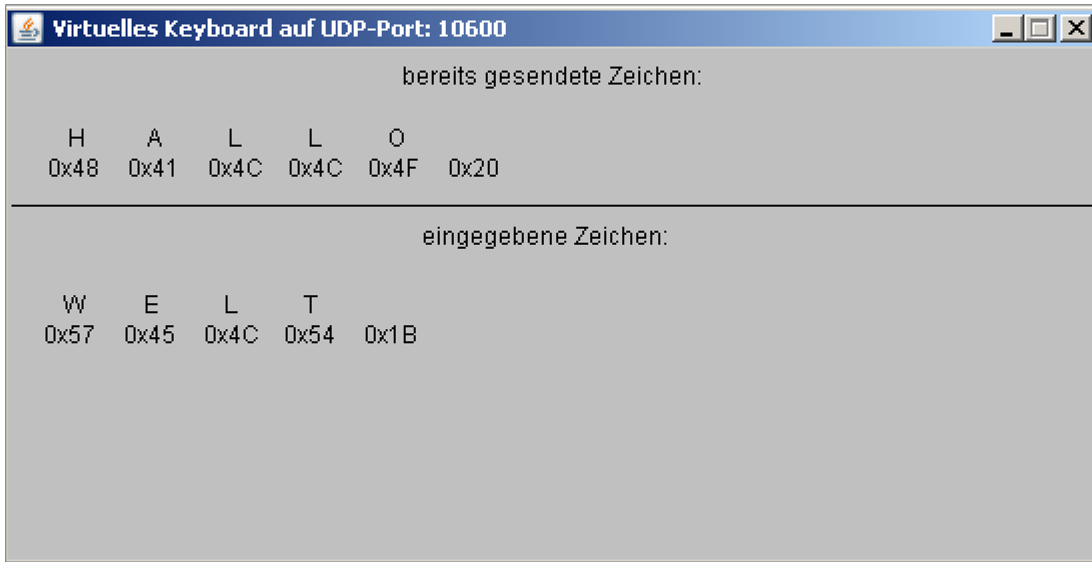


Abbildung 4-9: Keyboard-Oberfläche während der Simulation

Die allerletzte Flanke des **krequest_s-**Signals, im VHDL-Quellcode, vor dem Simulationsende, dient lediglich dazu den roten Anfrage-Schriftzug nachträglich auf der Keyboard-Oberfläche erscheinen zu lassen (Abbildung 4-10).

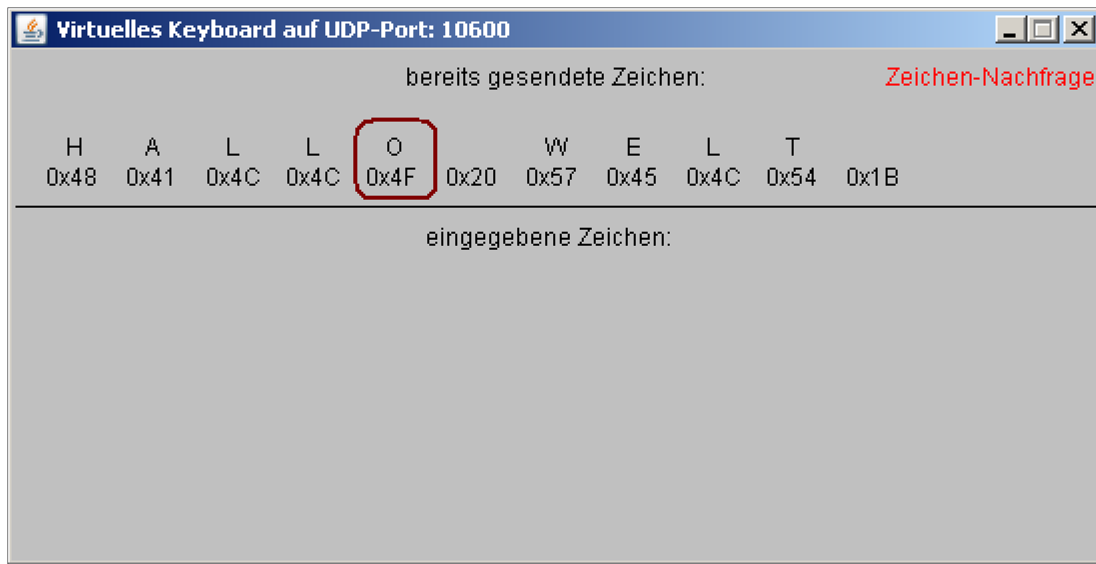


Abbildung 4-10: Keyboard-Oberfläche nach der Simulation

In dem Simulationsergebnis (Abbildung 4-11) ist der Empfang der Zeichen zu beobachten. Dabei sind die ASCII-Codes der übertragenden Zeichen sichtbar (vergleiche makiertes Zeichen 'O' mit ASCII-Code 0x4F)

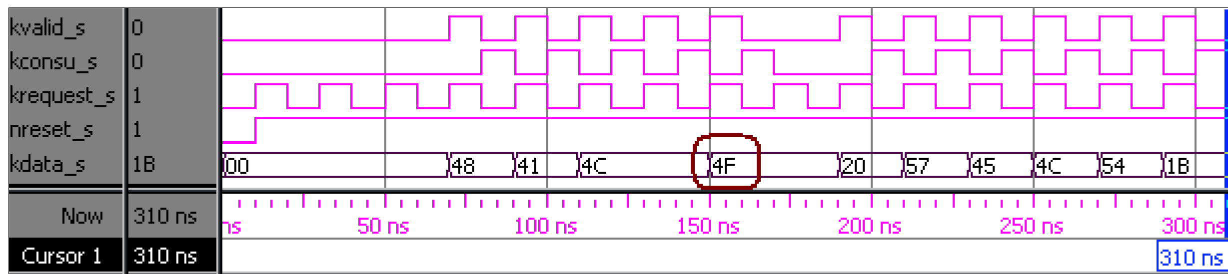


Abbildung 4-11: Simulationsergebnis vom virtuellen Keyboard

Die gesamte Dauer dieser Simulation, sowie die zum Teil unterschiedlichen zeitlichen Abstände zwischen den neuen Zeichen, sind von der Tastenanschlags-Frequenz und der eingestellten maximalen Wartezeit abhängig. Gibt man die gesamte Zeichenkette bereits vor der Simulation ein (siehe Abbildung 4-7), so wären die Abstände alle gleich lang und nur noch von den WAIT FOR - Anweisungen abhängig.

5 Kopplung von Simulationen

Bisher lag der Fokus vor allem darin, die ModelSim-Simulationen zu erweitern, um neue Möglichkeiten zur Kontrolle und Verifikation zu schaffen. In diesem Kapitel werden Ansätze diskutiert, wie mit den vorgestellten Methoden, die SystemC bietet, eine Kopplung von Simulationen realisiert werden kann. Im Vordergrund steht dabei die Motivation, die Simulationslast auf mehrere Rechner zu verteilen.

Da ModelSim nicht für verteilte Simulationen konzipiert wurde, müssen der Transport der verbindenden Signale sowie die Synchronisation zwischen den Simulationen von den SystemC-Schnittstellen übernommen werden. Der zentrale Punkt in diesem Unterfangen besteht darin, dem ModelSim-eigenen Scheduler im richtigen Zeitpunkt die Kontrolle über die Simulation zu entziehen und sie anschließend, nach den Signalübertragungen, wieder zurückzugeben. Dabei gilt es, die geschaffene Verbindung zu externen Prozessen vor dem Scheduler so zu verbergen, als würde sie gar nicht existieren. Im hier verfolgten Ansatz wird die Kommunikation zwischen den SystemC-Schnittstellen, welche die Simulationen verbinden, mit der Socket-API implementiert. Denn durch den Aufruf der blockierenden Empfangsfunktionen kann die laufende Simulation präzise angehalten und fortgesetzt werden.

Bei der Wahl der Synchronisationsformen zwischen den zu koppelnden Simulationen, sind verschiedene Varianten möglich. Die konkrete Auswahl ist im wesentlichen vom vorliegenden Anwendungsfall abhängig. Dabei stehen zwei Rahmenbedingungen im Vordergrund. Zum einen gilt es, die verfügbare Rechenleistung der beteiligten Rechner möglichst maximal auszunutzen, und zum anderen die Kopplungslogik so einfach wie möglich zu halten. Eine Verletzung der ersten Bedingung vermindert den Performance-Vorteil, der sich aus einer Lastverteilung ergibt. Dagegen führt eine zu 'intelligente' Schnittstellenlogik zu einem größerem Kommunikationsoverhead und lässt das Verhalten des Systems während der Simulation möglicherweise von dem Verhalten der realen Hardware-Implementierung abweichen. Eine theoretisch maximale Performance, bei der die Simulationen vollständig asynchron ohne Wartezeiten miteinander kommunizieren, ist nicht zu erreichen. Dies wurde bereits im vorigen Kapitel, bezüglich des Timings des virtuellen Keyboards, deutlich. Grund hierfür ist der ModelSim-interne 64Bit-Zeitzähler, der nach Start immer voranschreitet sobald zum aktuellen Zeitpunkt der Simulation keine Signaländerungen mehr zu berechnen sind. Das Problem besteht nun darin, dass externe Signale, die über die Grenzen einer abgeschlossenen, lokalen Simulation hinausgehen, in ModelSim nicht vorgesehen sind. Folglich gehen die Uhren selbst von zwei identischen, parallel laufenden Simulation niemals exakt gleich. Auch wenn der Startzeitpunkt bestmöglich synchronisiert wäre, käme es aufgrund von Hintergrundprozessen auf dem Betriebssystem zu Schwankungen. Oftmals sind die zu verbindenden Simulation auch nicht identisch und die physikalische Datenübertragung benötigt ebenfalls Zeit. Daher wären solche Simulationen komplett asynchron, also ohne die Kontrolle über die Simulationszeit abzugeben, nicht sinnvoll zu koppeln.

Nachfolgend werden mögliche Kopplungsformen mit unterschiedlichen Synchronisationsgraden anhand von allgemeinen Anwendungsfällen betrachtet. Anschließend folgt die Vorstellung und Bewertung von Implementierungsansätzen, mit denen sich die Kopplungen umsetzen lassen.

5.1 Vollsynchrone Kopplung

Diese Art der Kopplung eignet sich besonders für Systeme, die einer m-Server zu n-Client Struktur

genügen. Zum Beispiel könnte eine Mehrkern-Motherboard-Simulation ihre CPUs als eigenständige Simulationen parallel mit Arbeit beauftragen. Weiterhin ist es mit dieser Variante sehr einfach Teilkomponenten, zum Beispiel aus Gründen der Speicherverfügung, auf andere Rechner auszulagern.

Der Kontrollfluss kann an beliebige Signalpegel gebunden werden und wird im Synchronisationsmoment von einer Simulation vollständig an eine andere abgegeben. Dabei blockieren die Client-Simulationen bis die entsprechenden Server-Simulationen ihre Ergebnisse abgeliefert haben. Im Normalfall überträgt eine Client-Simulation ihre Ausgangssignalpegel an die Server-Simulation und initiiert damit die Kommunikation. Daraufhin arbeitet die Server-Simulation mit den übernommenen Signalzuständen und stellt ihre Ergebnisse dem Auftragsgeber anschließend zur Verfügung. Hierzu eine Beispielanwendung (Abbildung 5-2).

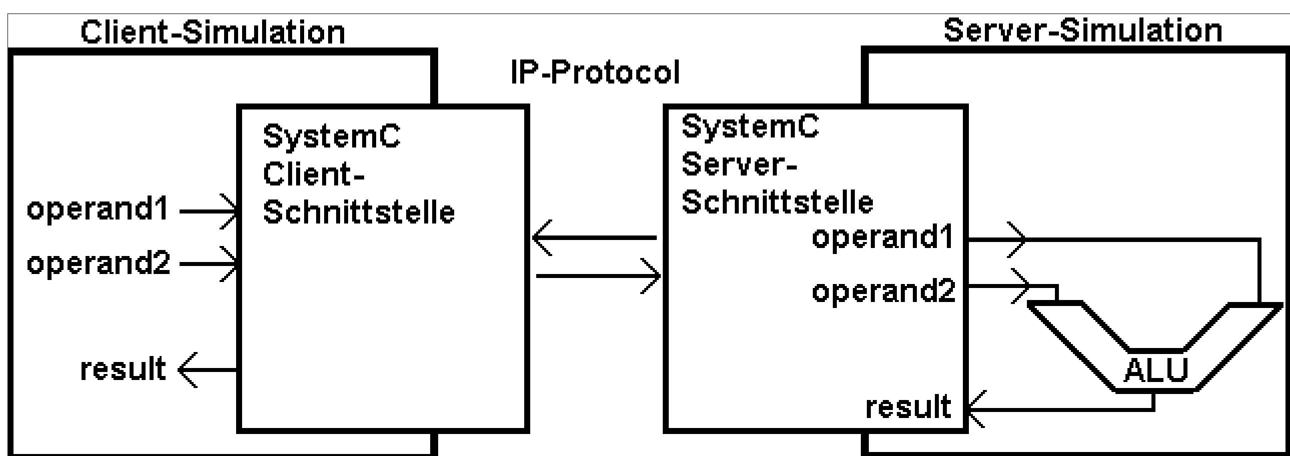


Abbildung 5-1: Beispiel einer vollsynchronen Kopplung

Bei der Client-Simulation handelt es sich um einen beliebigen Prozess, der zu irgendeinen Zeitpunkt einen Prozess in der Server-Simulation mit Berechnungen beauftragen möchte. Hier sollen beispielhaft zwei Signale (operand1, operand2), die von Client-Simulation erzeugt werden, in der Server-Simulation zu einem Signal (result) verknüpft werden. Dieses Ergebnis wird anschließend, aus Sicht der aktiven Simulation innerhalb eines Delta-Zyklus, zurück übertragen. Der Kontrollfluss verhält sich dabei wie folgt:

1. Die clientseitige VHDL-Simulation legt die Parametersignale zum Auftrag an ihre SystemC-Schnittstelle an.

2. In dem entsprechenden SystemC-Prozess werden die angelegten Signale per Socket-API Sendefunktion an den Rechner, auf dem die serverseitige Simulation läuft, gesendet. Unmittelbar danach wartet der SystemC-Prozess durch den Aufruf einer blockierenden Empfangsfunktion und pausiert damit die eigene Simulation.

3. Die SystemC-Serverschnittstelle hat die Server-Simulation bereits seit dem Start der Simulation, in ihrem Initialisierungsprozess, in einen empfangs-blockierenden Zustand versetzt. Dieser Prozess empfängt nun die Daten von der Client-Schnittstelle und aktualisiert die entsprechenden Signale.

4. Durch die Aktualisierung wird der ALU-Prozess, der pegelsensitiv auf die Parameter-Signale reagiert, angestoßen und setzt die VHDL-Simulation fort. Nachdem der Prozess

seine Berechnungen abgeschlossen hat, aktualisiert er das Ergebnissignal.

5. Ein, auf das Ergebnissignal pegelsensitiver, SystemC-Prozess sendet das Ergebnis an den Auftraggeber zurück und versetzt die Simulation wieder in den empfangs-blockierenden Zustand, so dass die Simulation neue Aufträge annehmen kann.

6. Nachdem das Ergebnis bei der Client-Schnittstelle eingetroffen ist, wird die VHDL-Simulation mit dem neuen, aktualisierten Ergebnis fortgesetzt.

Dieser gesamte Ablauf geschieht aus Sicht der Client-Simulation in einem Delta-Zyklus.

Vorteilhaft bei dieser Kopplungsform ist, dass die SystemC-Schnittstellen direkt miteinander Kommunizieren. Dadurch ist dieses synchrone Verfahren einfach zu implementieren und sehr skalierbar. Damit eine passive Simulation einen Arbeitsauftrag empfangen kann, muss der Auftraggeber ihr nicht explizit bekannt sein, da er sich beim Versenden der Datenpakete automatisch als Absender durch seine IP-Adresse und Port identifiziert. Die Server-Simulation verhält sich nach ihrem Start somit vollständig reaktiv und kann alle eintreffenden Aufträge von mehreren Client-Simulationen sukzessiv abarbeiten. Auf der anderen Seite kann eine Client-Simulation beliebig viele Verbindungen parallel aufbauen.

5.2 Taktsynchrone Kopplung

Im Gegensatz zur vollsynchronen Kopplung unterscheidet sich dieser Variante dahin gehend, dass stets ein Taktsignal für die Synchronisierung genutzt wird.

Besteht ein System aus Komponenten, die ihre Signale synchron zu einem Takt aktualisieren, kann eine der beteiligten Simulationen als Taktgeber fungieren. Als ein anschauliches Beispielszenario für taktsynchrone Systeme, soll hier die verteilte Kopplung von einzelnen Pipelinestufen dienen (Abbildung 5-1).

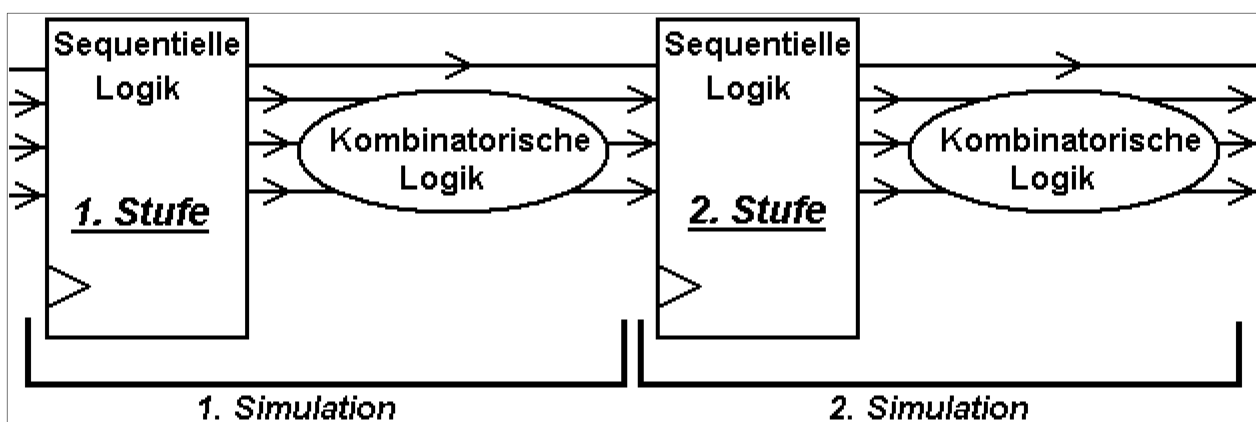


Abbildung 5-2: Beispielhafte 2-stufigePipeline

Dabei sendet eine der Stufen ein Taktsignal an alle Anderen. Nach dem Empfang des Taktes und Beendigung der jeweiligen Berechnungen, schickt jede der anderen Stufen ihre Ausgangssignale an die jeweils nächste Stufe. Unmittelbar danach sendet jede Komponente ein zusätzliches Bestätigungspaket an den Taktgeber, der nach dem Erhalt aller Bestätigungen mit dem nächsten Taktzyklus beginnen kann.

Entsprechend der verwendeten Netzwerktopologie könnte es passieren, dass eine der Simulationen das Datenpaket der vorigen Stufe zeitlich vor dem Taktsignal des Taktgebers erhält. Die zuerst ankommenden Datensignale könnten, anstelle des eigentlichen Taktsignals, als Takt interpretiert werden. Ein nachträglich empfangendes (echtes) Taktsignal kann dann verworfen werden. Alternativ müsste die SystemC-Schnittstelle, je nach Anwendungsfall, eine Pufferlogik realisieren, die eine ausreichende Wahrung der Kausalordnung (*Happened-Before-Relation*) garantiert.

5.3 Halbsynchrone Kopplung

Sollte die Einteilung von Hardware-Komponenten in Client und Server nicht möglich sein oder sind länger pausierte Simulationen aus Effizienzgründen nicht hinnehmbar, bedarf es einer Lockerung der Synchronisation. Besonders für homogene Simulationen, die bidirektional Signale austauschen ist dieser Umstand gegeben.

Als Lösungsansatz kommt eine gepufferte Kopplung in Frage. Dabei läuft jede Simulation für sich erst einmal auf voller Last. Währenddessen wird jede Signaländerung, die für eine parallel laufende Simulation interessant ist, gepuffert und an diese übertragen. Bei jedem dieser Sendevorgänge wird das Datenpaket, mit den aktuellen Signalwerten, in eine Warteschlange für die jeweils andere Simulation, zwischengespeichert. Erst wenn eine der beiden Warteschlangen voll ist, also einen definierten Füllstand überschritten hat, blockiert die entsprechende Simulation bis die andere Simulation wieder Platz geschaffen hat. So ist sichergestellt, dass die Simulationen keine Signalaktualisierungen verpassen.

Die beschriebene Pufferfunktionalität erfordert eine Softwarekomponente, die zwischen je zwei Simulationen koordiniert. Einzige Anforderung an die Software ist dabei, dass sie ausschließlich das Timing der Pakete steuert, ohne jede weitere Form von Intelligenz, um nicht die Synthesefähigkeit einzubüßen. Je nach Anwendungsfall ist anstelle einer Paketobergrenze auch eine maximale Zeitdifferenz als Höchstgrenze für das Auseinanderdriften denkbar. Dies ist insbesondere dann sinnvoll, wenn die Synchronität von Zeitstempeln und nicht von einer Anzahl an Signalaktualisierungen abhängig sein soll. Im Zweifelsfall ist auch eine Kombination beider Kriterien problemlos möglich.

Zur Demonstration einer gepufferten Kopplung dient ein überschaubares Beispielszenario , in dem zwei identische Simulationen bidirektional miteinander kommunizieren (Abbildung 5-3).

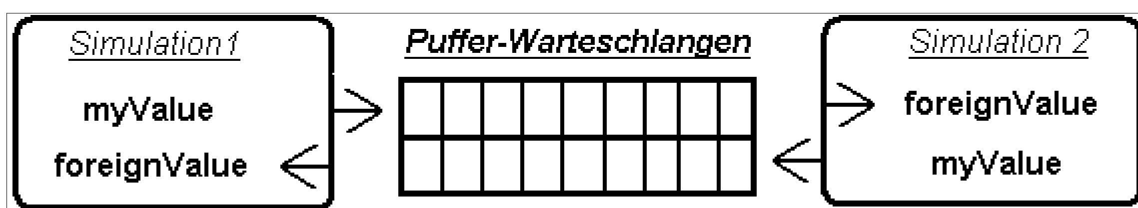


Abbildung 5-3: Darstellung der bidirektionalen Kopplung

Beide Simulationen verfügen über die gleiche SystemC-Schnittstelle und haben jeweils ihren eigenen Takt. In jeder Simulation existieren genau zwei 32Bit-Signale (Abbildung 5-4). Der gesamte Quellcode zu diesem Beispiel befindet sich ebenfalls auf der CD zu dieser Arbeit.

```
21 SIGNAL myValue_s, foreignValue_s : std_logic_vector(31 DOWNTO 0);
22
23 COMPONENT bidirectional_interface -- IP-Adresse und UDP-Port
24    GENERIC ( ip_addr1, ip_addr2, ip_addr3, ip_addr4, udp_port : integer );
25    PORT (
26       myValue_p      : IN std_logic_vector(31 DOWNTO 0);    -- Eigener Wert
27       foreignValue_p : OUT std_logic_vector(31 DOWNTO 0) ); -- Fremder Wert
28 END COMPONENT bidirectional_interface;
29
30 BEGIN
31    my_bidirectional : bidirectional_interface
32    GENERIC MAP (127, 0, 0, 1, 10200 ) -- Schnittstelle einbinden
33    PORT MAP ( myValue_s, foreignValue_s);
34
35     test_process : PROCESS
36     BEGIN -- Prozess mit eigenem Takt
37         myValue_s <= x"00000000"; -- Eigenen Datenwert initialisieren
38         WAIT FOR 10 ns;
39         WHILE ( foreignValue_s < x"00010000" ) LOOP -- Ende-Bedingung prüfen
40             myValue_s <= myValue_s + 1; -- Signaländerung
41             WAIT FOR 10 ns;
42         END LOOP;
43         WAIT; -- Simulationsende
44     END PROCESS;
```

Abbildung 5-4: Testbench für bidirektionale Kopplung

In einem VHDL-Prozess wird stets eins der Signale in einer Schleife inkrementiert (Zeile 40) und durch diese Veränderung automatisch an die korrespondierende Simulation übermittelt. Während dieses Signal (*myValue_s*) also die Durchläufe der eigenen Simulation zählt, entspricht das andere Signal (*foreinValue_s*) den Durchläufen der anderen Simulation. Die Simulation endet, sobald der Zählerstand, der jeweils anderen Simulation, einen festen Wert erreicht hat (Zeile 39). Analog zur virtuellen Konsole, werden den Simulationen IP-Adresse und Portnummer der Koordinierungssoftware, über die generischen Parameter beim Einbinden der Schnittstelle mitgeteilt (Zeile 32).

Hier ist die Koordinierungssoftware in Java geschrieben, kann aber ebenso gut, sofern die Socket-API unterstützt wird, in jeder beliebigen Programmiersprache implementiert werden. Vor dem Simulationsstart muss die Software bereits gestartet sein und kann dann auch zwischen Simulationszyklen ohne Neustart wiederverwendet werden. Soll vollständig auf eine zusätzliche Software verzichtet werden, ist es alternativ auch möglich, die Pufferlogik direkt in der SystemC-Schnittstellen zu realisieren.

Die Simulationen selber können stets nacheinander gestartet werden und synchronisieren sich über die Koordinierungssoftware einmalig zu Beginn, so dass sie zum selben Zeitpunkt loslaufen. Als Kommandozeilenparameter werden in diesem Szenario die UDP-Portnummer zusammen mit der minimalen und maximalen Puffergröße in Bytes übergeben.

Abgesendete Datenpakete werden von der Software sofort beantwortet, damit die sendende Simulation nicht weiter blockiert. Nur wenn die jeweilige Sender-Warteschlange bereits voll sein sollte, wird die Simulation erst dann durch eine Antwort aus der Empfangsblockierung gelöst, wenn die jeweils andere Simulation einen neuen Signalwert übertragen möchte und dabei gleichzeitig

durch das Abholen eines Paketes, einen Platz freigibt. Die Angabe der minimalen Puffergröße dient im wesentlichen dazu einen Pufferunterlauf zu verhindern. So kann im Falle einer eigentlich leeren Schlange der letzte, bereits abgeholte Wert, wieder als aktuell gelesen werden. Durch diese Mindestgröße entfallen die Abfragen für eine zusätzliche Fallunterscheidung. Im Regelfall dürfte die Mindestgröße genau einer Paketgröße entsprechen.

Nachfolgend soll das vorgestellte Beispielszenario demonstriert werden.

Im ersten Versuch beträgt die maximale Puffergröße 8 Bytes und da bei jeder Übertragung ein 32Bit-Signal übertragen werden soll, wurde eine minimale Puffergröße von 4 Bytes verwendet. Die Simulationen können so um effektiv 8 – 4 = 4 Bytes, also maximal einer Signaländerung, auseinanderdriften. Das Ergebnis der Simulation (Abbildung 5-5) zeigt die langsamere Simulation (oben) im Vergleich zur schnelleren Simulation (unten).

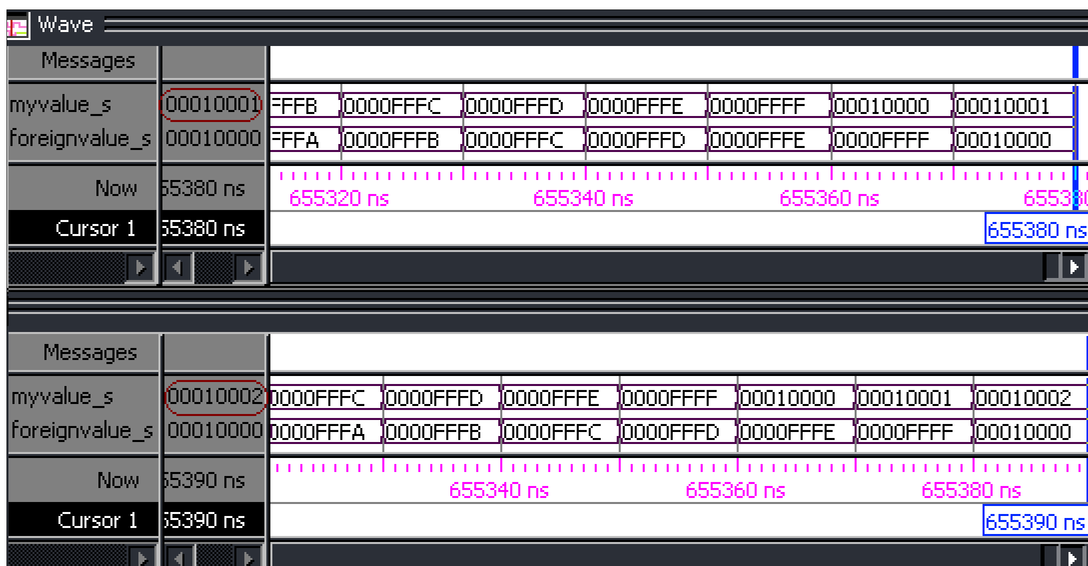


Abbildung 5-5: Simulationsergebnis mit 8-Byte-Puffergröße

Jede der beiden Simulationen wurde beendet, sobald sie jeweils den Wert *0x10000 = 65.536* aus ihrer Warteschlange empfangen hat. Wie erwartet liegen die beiden Simulationen am Ende um einen Durchlauf bzw. 10 Nanosekunden Simulationszeit auseinander.

Anschließend lässt sich der zeitliche Drift in Abhängigkeit der Puffergröße, in einem zweiten Versuch, diesmal mit einer maximalen Puffergröße von 400 Bytes, untersuchen (Abbildung 5-6).

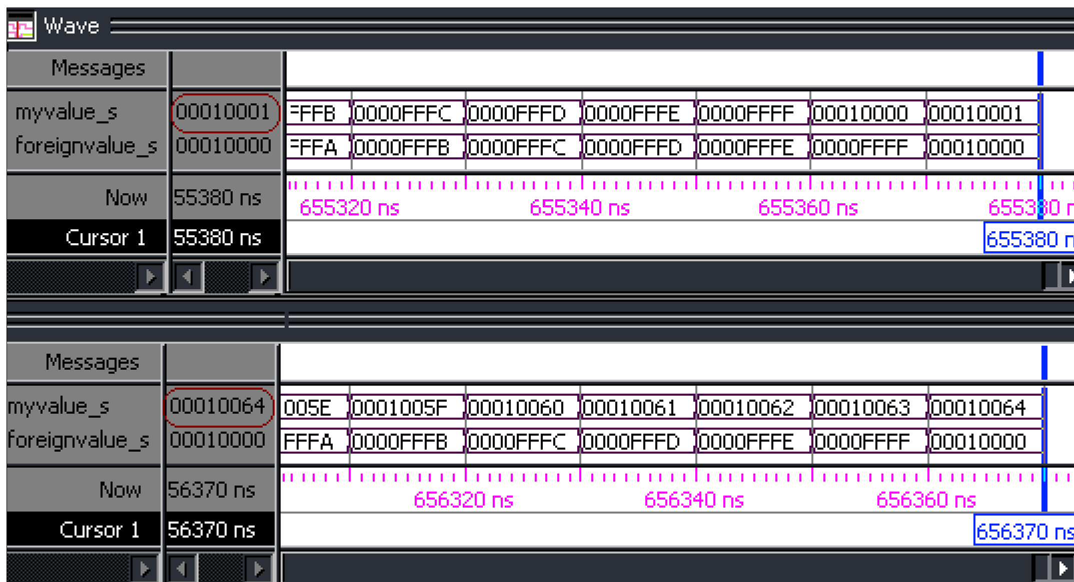


Abbildung 5-6: Simulationsergebnis mit 400-Byte-Puffergröße

Diesmal ist rechnerisch ein maximaler Abstand von 400 – 4 = 396 Bytes, also 396 / 4 = 99 = 0x63 Signalaktualisierungen zu erwarten. Diese Vermutung konnte vom Simulationsergebnis bestätigt werden.

6 Fazit

In dieser Arbeit wurden verschiedene Aspekte und Möglichkeiten, welche SystemC zur Erweiterung von ModelSim bietet, untersucht und demonstriert. Ohne den Einsatz von SystemC-Schnittstellen werden Einfluss und Kontrolle über die laufende ModelSim Simulation nur sehr rudimentär unterstützt. Weiterhin fehlt es an Möglichkeiten Teilkomponenten zu virtualisieren oder mehrere Simulationen – zwecks Lastverteilung – zu koppeln. Vorgestellt wurden unterschiedliche Schemata zum Aufbau von SystemC-Schnittstellen, mit denen die genannten Erweiterungen einfach realisiert werden können. Durch den Einsatz von SystemC-Schnittstellen kann eine Verbindung von Hardwarebeschreibungs- zu Programmiersprachen und zurück realisiert werden. Der Entwickler hat nun die Möglichkeit, wahlweise Funktionalitäten von Hardwarekomponenten durch den Einsatz von C/C++ zu implementieren. Dies ermöglicht insbesondere die zusätzliche Verwendung der Socket-API, die SystemC unterstützt, mit der ie VHDL-Simulation unter ModelSim, über ihre Grenzen hinweg, erweitert werden kann. Aus diesem breiten Spektrum von potentiellen Erweiterungsmöglichkeiten sind ein 640x480-Monochrom-Display sowie ein virtuelles Keyboard als fertige Anwendungen im Rahmen dieser Arbeit entstanden.

Insbesondere bei schwer zu simulierenden Systemen, wäre die Entschärfung von zahlreichen Problemstellungen durch Virtualisierung denkbar. Die virtuellen Freiheitsgrade werden jedoch durch einige vorhanden Grenzen eingeschränkt, denn der native ModelSim-eigene Scheduler wurde nicht für eine Kommunikation über seine Grenzen hinweg konzipiert. Erforderlich und zwingend notwendig zur Umgehung dieser Grenzen, ist das Aufrufen der blockierenden C/C++ Empfangsfunktionen, mit denen die Simulation angehalten werden kann, so dass eine Synchronisation an relevanten Zeitpunkten garantiert wird. Darüber hinaus ist bei der Auslegung der Logik in den Softwarekomponenten besondere Sorgfalt geboten. Denn um die Synthetisierbarkeit des virtuellen Systems zu gewährleisten, muss für die verwendete Intelligenz ein Gegenstück in Hardware existieren.

Noch offen bleibt die Frage nach der praktischen Handhabung von gekoppelten Simulationen im realen Industrieumfeld. Eine Untersuchung des Aufwands für den Einsatz von SystemC-Schnittstellen sowie Performance-Analysen an konkreten Systemen könnten durchaus, im Rahmen einer zukünftigen Arbeit, ergänzt werden.

Anhang A – Anwendungsfall: LED-Lauflicht

Die Virtualisierung bietet sich zur Verifizierung von Hardware besonders dann an, wenn eine fehlerhafte Komponente zur Zerstörung von Bauteilen führen kann. In einem möglichen Laborversuch zur VHDL-Programmierung soll es Aufgabe sein ein LED-Lauflicht, bestehend aus 8 CMOS-Halbleiterdioden, korrekt zu beschalten. Wenn die LED leuchtet liegt die Kathode auf Masse und die Anode auf einem High-Pegel. Sollten fehlerhafte oder hochohmige Pegel anliegen, ist das Verhalten der Diode undefiniert und kann unter Umständen zu einem Defekt führen. Die 4-wertigen Logikvektoren, die SystemC unterstützt, können zu Fallunterscheidung (Tabelle 6-1) der angelegten Pegel genutzt werden.

Kathoden-Pegel	Anoden-Pegel	LED-Verhalten
0	1	angeschaltet
0	0	ausgeschaltet
1	1	ausgeschaltet
1	0	falsch beschaltet
Z / X	don't care	falsch beschaltet
don't care	Z / X	falsch beschaltet

Tabelle 6-1: LED-Beschaltungsvariationen

Die SystemC-Schnittstelle stellt dem VHDL-Entwickler das LED-Lauflicht als eigenständige Komponente zur Verfügung (Abbildung 6-1).

```
COMPONENT LED_interface
   GENERIC ( ip_addr1, ip_addr2, ip_addr3, ip_addr4, udp_port : integer );
   PORT (
        data_p, ground_p : IN std_logic_vector(7 DOWNTO 0);
        nReset_p         : IN std_logic);
END COMPONENT LED_interface;
```

Abbildung 6-1: Schnittstelle zum LED-Lauflicht

Ein pegelsensitiver SystemC-Prozess übermittelt den Zustand an die korrespondierende Java-Anwendung, welche die LED-Zustände visualisiert. Wie bei der virtuellen Konsole werden IP-Adresse und UDP-Port als generische Parameter übergeben. Ein Low-Pegel auf der Reset-Leitung repariert defekte LEDs wieder. Zur Überprüfung der Funktionsweise dient eine Testumgebung (Abbildung 6-2), in der einige LEDs korrekt angeschaltet werden und eine falsch beschaltet wird.

Zur Übertragung der Zustände reichen zwei Bytes. Ein Byte enthält die Information, welche LEDs richtig oder falsch angeschlossen sind und das zweite, ob eine korrekt angeschlossene Diode an- oder ausgeschaltet ist.

```
-- LED-Lauflicht einbinden
my_LED : LED_interface
GENERIC MAP (127, 0, 0, 1, 10700 ) -- IP, Port
PORT MAP ( data_s, ground_s, nReset_s );

test_process : PROCESS
BEGIN
   -- Initialisierungen
   data_s <= (OTHERS => '0');
   ground_s <= (OTHERS => '0');
   nReset_s <= '0';
   WAIT FOR 10 ns;
   nReset_s <= '1';
   data_s <= x"2B";   -- Einige LEDs anschalten
   ground_s <= x"80"; -- Eine LED falsch beschaltet
   WAIT FOR 10 ns;
   ground_s <= x"00";
   WAIT;
END PROCESS;
```

Abbildung 6-2: Testbench zum LED-Lauflicht

Unmittelbar vor dem Ende der Simulation wird die falsche LED-Beschaltung wieder korrigiert. Dennoch bleibt die einmal falsch angeschlossene Leuchtdiode bis zum nächsten Reset defekt, was nach der Simulation auf der Java-Oberfläche (Abbildung 6-3) zu erkennen ist.

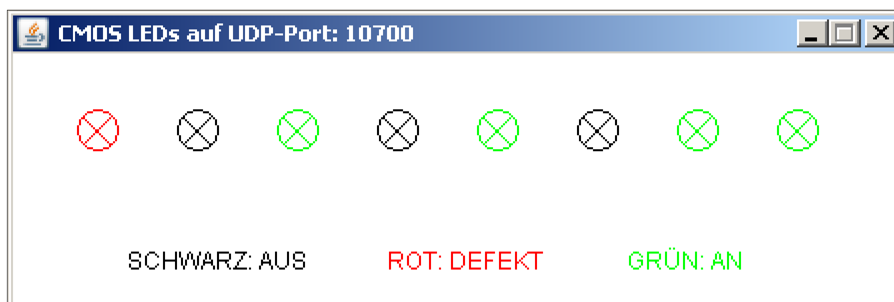


Abbildung 6-3: Lauflicht-Oberfläche nach der Simulation

Anhang B – Inhalt der CD

Neben dieser Arbeit im PDF-Format, befinden sich im Wurzelverzeichnis der CD folgende Ordner:

DFlipFlop: TCL-Skript, VHDL- und SystemC-Quellcode zur Demonstration der Implementierung eines D-FlipFlops in SystemC zu Kapitel 2.2.

Virtual_RAM: TCL-Skript, VHDL-, SystemC- und Java-Quellcode mit ausführbaren Java-Bytecode für den virtuellen Videospeicher zu Kapitel 3.6.

Virtual_Keyboard: TCL-Skript, VHDL-, SystemC- und Java-Quellcode mit ausführbaren Java-Bytecode für das virtuelle Keyboard zu Kapitel 4.5.

Simulations_Kopplung: TCL-Skript, VHDL-, SystemC- und Java-Quellcode mit ausführbaren Java-Bytecode für die Kopplung zweier Simulationen zu Kapitel 5.3.

LED_Lauflicht: TCL-Skript, VHDL-, SystemC- und Java-Quellcode mit ausführbaren Java-Bytecode für das CMOS-LED-Lauflicht im Anhang A.

Quellen: Vergängliche Quellen aus dem Literaturverzeichnis.

Glossar

ASCII: Abkürzung für *American Standard Code for Information Interchange.* Eine normierte 7-Bit Zeichenkodierung, die aus 33 nicht-druckbaren, sowie 95 druckbaren Zeichen besteht. Sie umfasst im Wesentlichen das lateinische Alphabet, die arabischen Ziffern und einige satz- und Steuerzeichen. Damit deckt der Zeichenvorrat dem einer Tastatur weitgehend ab.

Delta-Zyklus: Imaginäre Zeiteinheit, in der VHDL-Signalzuweisungen gesammelt und aktualisiert werden, die dazu dient die Nebenläufigkeit von Hardware zu simulieren [14].

Alignment: Eine Dateneinheit gilt als ausgerichtet, wenn ihre Adresse im Speicher ein ganzzahliges Vielfaches einer Zweierpotenz ist.

Java: Objektorientierte und plattformunabhängige Programmiersprache der Firma Sun Microsystems, die im Jahr 1995 erschien. Java-Programme laufen auf allen Betriebssystemen, für die eine spezielle Laufzeitumgebung, die sogenannte *Java Virtual Machine*, existiert.

Little Endian: Byte-Reihenfolge, bei der das niederwertigste Byte eines Wortes bzw. das niederwertigste Bit eines Bytes auf der kleineren Adresse im Speicher abgelegt wird.

ModelSim: Simulationsprogramm der Firma Mentor Graphics, für die rechnerunterstützte Entwicklung von elektronischen Schaltungen mit Hardwarebeschreibungssprachen wie VHDL oder Verilog.

Socket: Abgeleitet vom englischen *Sockel.* Softwareschnittstelle vom Betriebssystem, die dem Programmierer das Senden und Empfangen von IP-Paketen und somit eine (ggf. verteilte) Interprozesskommunikation ermöglicht.

SystemC: C++ Klassenbibliothek zur Modellierung von elektronischen Systemen auf hohem Abstraktionsgrad, die sowohl Hardware- als auch Softwarekomponenten enthalten. SystemC bietet darüber hinaus die Möglichkeit VHDL um zusätzliche Funktionalitäten zu erweitern.

TCL: Abkürzung für *Tool Command Language*. Universelle und plattformunabhängige Makrosprache, mit dem Ziel außerordentlich einfach gestrickt zu sein. ModelSim hat die Möglichkeit TCL-Skripte auszuführen und somit die Befehlsausführung zu automatisieren.

UDP: Abkürzung für *User Datagram Protocol*. Verbindungsloses, ungesichertes und paketorientiertes Transport-Protokoll , welches rohe IP-Pakete mit einem Header hauptsächlich um Quell- und Zielportnummern erweitert, so dass eine Zuordnung der Pakete zu Prozessen auf Betriebssystemebene ermöglicht wird.

VGA: Abkürzung für *Video Graphics Array.* Dieser analoge Bildübertragungsstandart von IBM wird für Stecker- und Kabelverbindungen zwischen Grafikkarten und Anzeigegeräte verwendet.

VHDL: Abkürzung für *Very High Speed Integrated Circuit Hardware Description Language*. Dabei handelt es sich um eine hierarchische, mit einer Programmiersprache vergleichbare Hardwarebeschreibungssprache, die zur Entwicklung komplexer, digitaler Schaltungen und Systeme dient. Seit ihrer Entstehung in den frühen 1980er Jahren hat sich VHDL neben Verilog, insbesondere im europäischen Raum als Industriestandart etabliert.

Literaturverzeichnis

1: David C. Black and Jack Donovan, SystemC: From The Ground Up, Springer-Verlag, 1. Auflage 2004

2: Prof. Jürgen Reichardt und Bernd Schwarz, VHDL-Synthese, Oldenbourg Wissenschaftsverlag GmbH, 4. Auflage 2007

3: Prof. Michael Schäfers, Rechnerstrukturen Vorlesungsfolien HAW Hamburg, Sommersemester 2009

4: Request for Comments: 3330, http://tools.ietf.org/html/rfc3330, Zugriff: 18. Februar 2010

5: Request for Comments: 4291, http://tools.ietf.org/html/rfc4291, Zugriff: 18. Februar 2010

6: IEEE 1666 - SystemC LRM, http://standards.ieee.org/getieee/1666, Zugriff: 19. Februar 2010

7: MinGW, http://www.mingw.org, Zugriff: 19. Februar 2010

8: SystemC-Dokumentation, http://www.lysium.de/docs, Zugriff: 23. Februar 2010

9: Prof. Michael Schäfers, SystemC-WP Vorlesungsfolien HAW Hamburg, Sommersemester 2007

10: Mentor Graphics, ModelSim SE User's Manual Kapitel 6 - SystemC Simulation, UM-158 2006

11: Volker Claus und Andreas Schwill , Duden Informatik, 1. Auflage 2003

12: Steve Furber, ARM System-on-Chip Architecture, 2.Auflage 2000

13: Microsoft Developer Network, http://msdn.microsoft.com/en-us/library/ms740673%28VS.85%29.aspx, Zugriff: 15. März 2010

14: Sun Microsystems, Inc., http://java.sun.com/javase/6/docs/api/, Zugriff: 31.03.2010

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort, Datum Unterschrift