TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Einsatz von SystemC im Hardware/Software-Codesign

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Karl Riedling
und Dipl.-Ing. Dr.techn. Gerhard R. Cadek

E 366
Institut für Industrielle Elektronik und Materialwissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik

von

Hannes Muhr
Matrikelnummer 9525319
Hauptstraße 58, 2463 Gallbrunn

Wien, im November 2000 Unterschrift

Diese Arbeit wurde am Entwicklungszentrum für Elektronik bei Siemens in Wien durchgeführt.

Betreuung bei Siemens:

Abteilung PSE ECT Technologiezentrum Systemverifikation
Dipl.-Ing. Johann Notbauer
und Dipl.-Ing. Dr. Georg Niedrist

An dieser Stelle möchte ich mich bei jenen Personen bedanken, mit deren Hilfe und Unterstützung diese Arbeit zustande gekommen ist.

Besonderer Dank gilt Herrn Dipl.-Ing. Johann Notbauer, der mir während meiner Ferialjobs bei Siemens sehr viel Wissen vermittelt hat, das nicht nur in dieser Arbeit anwendbar war.

Weiterer Dank gebührt Herrn Dr. Georg Niedrist, der die interessante Themenstellung und einen Arbeitsplatz bei Siemens ermöglichte.

Für die fachkompetente Betreuung möchte ich Herrn Dr. Gerhard Cadek danken, der mir wertvolle Hinweise zur Literatur gab und das zeitaufwendige Korrekturlesen durchführte.

Kurzfassung

Die Entwicklung von elektronischen Systemen erfordert ständig verbesserte Methoden und Werkzeuge, um die Anforderungen des Marktes erfüllen und dem Konkurrenzdruck standhalten zu können. Es werden deshalb laufend Überlegungen angestellt, wie man effektivere Methoden nutzen kann, die gleichzeitig eine Steigerung der Qualität der Ergebnisse bedeuten und die Wiederverwendung von bestehenden Systemen erleichtert.

In dieser Arbeit wird ein Ansatz namens SystemC untersucht, der die Programmiersprache C++ um Konstrukte zur Modellierung von Hardware erweitert, und so den Top-Down-Systementwurf von der Spezifikation bis hin zur implementierbaren Bauteilnetzliste unterstützt. Die Syntax von SystemC wird kurz umrissen und ein konkretes System implementiert, anhand dessen die Einbindung in den bestehenden Entwurfsablauf gezeigt wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

Bereits im Jahr 1965 formulierte Gordon Moore, ein späterer Mitbegründer des Chipherstellers Intel, dass sich die Integrationsdichte von integrierten Schaltungen etwa alle 18 Monate verdoppeln wird. Dies ermöglicht die Entwicklung immer komplexerer Systeme auf einem Chip. Besonders Firmen aus den Sektoren Multimedia, Spiele und Telekommunikation treiben diesen Fortschritt heute massiv voran. Der im Zusammenhang mit dem Funktionalitäts- und Komplexitätsanstieg entstandene Begriff *System-on-Chip* (kurz SoC) beschreibt integrierte Schaltkreise, die eine Vielfalt an Funktionalität aufweisen und dafür neben der Hardware für Schnittstellen und interner Logik auch Speicher und Prozessorkerne für die Ausführung von Software enthalten – eben gesamte Systeme.

Die derzeitigen SoC-Entwicklungs- und Verifikationsmethoden weisen im Allgemeinen eine relativ strikte Trennung zwischen Hardware und Software auf. Systemarchitekten entwerfen eine Spezifikation und übergeben diese an Hardware- und Software-Teams, die mit geringer Interaktion ihren jeweiligen Entwurf durchführen und verifizieren. Der Grund dafür liegt im Fehlen von Methoden, Werkzeugen oder einheitlichen Sprachen, die es erlauben, sowohl Softwarealgorithmen als auch Hardwarekomponenten zu beschreiben. Die Cosimulation von Hardware und Software sowie die schrittweise Verfeinerung des Systementwurfs hinunter zu seinen Bauteilen ist dadurch stark erschwert [1].

Eine Definition einer gänzlich neuen Sprache samt Werkzeugen könnte zwar auf diese Anforderungen maßgeschneidert werden, ihre wirtschaftliche Akzeptanz wäre aber fraglich. Neue Werkzeuge müssten erst entwickelt werden und der bereits bestehende Code im Hardware- und Softwarebereich könnte nicht weiter verwendet werden. Im Gegensatz dazu steht eine Erweiterung von bereits bestehenden Sprachen. Dieser Ansatz wird bei *SystemC* verfolgt. SystemC basiert auf der Programmiersprache C++, in die Konzepte zur Hardware-Modellierung wie Parallelität, Reaktionsfähigkeit auf Ereignisse, Hierarchiestrukturierung und Hardware-Datentypen eingebracht wurden.

In dieser Arbeit wird die Verwendbarkeit von SystemC in der Systementwicklung

anhand eines konkreten Beispiels aus dem Bereich der Telekommunikation untersucht. Das Ziel der Untersuchungen war, Aussagen über die Praxisrelevanz und die Anwendbarkeit der Sprache SystemC samt Werkzeugen zu treffen. Es sollte die Frage geklärt werden, ob damit wie versprochen eine neue, effizientere Methode für den Entwurf und die Verifikation von komplexen Hardware-Software-Systemen zur Verfügung steht.

Kapitel 2

Motivation und Aufgabenstellung

Traditionelle Entwurfsmethoden betrachten die Entwicklung von Hardware und Software getrennt voneinander. Die Software wird unter Voraussetzung einer detaillierten Spezifikation parallel zur Hardwareentwicklung implementiert. Der Test der Software kann deshalb erst mit der Verfügbarkeit von Hardwaremustern beginnen, wodurch Fehler in der Software sehr spät entdeckt werden. Insbesondere Fehler in der Hardware/Software-Schnittstelle führen zu zeit- und kostenintensiven Redesigns der Hardware. Entwicklungszyklen von 9 - 12 Monaten, wie sie derzeit u. a. im Telekomsektor üblich sind, könnten mit dieser Methodik nicht oder nur mit großem organisatorischem Aufwand erreicht werden.

Dem Wunsch folgend, Hard- und Software bereits in einem früheren Stadium der Entwicklung gemeinsam zu verifizieren, entstand die Hardware/Software-Coverifikation (siehe Abbildung 2.1). Der Software wird dadurch eine virtuelle Hardwareplattform, etwa durch ein Simulationsmodell, zur Verfügung gestellt. Die Time-to-Market konnte damit deutlich gesenkt und die Qualität der Hard- und Software erhöht werden [2, 3]. Der Nachteil dabei liegt im zusätzlichen Aufwand für die Modellierung der Hardware/Software-Schnittstelle. Die Kommunikation zwischen der Simulation der

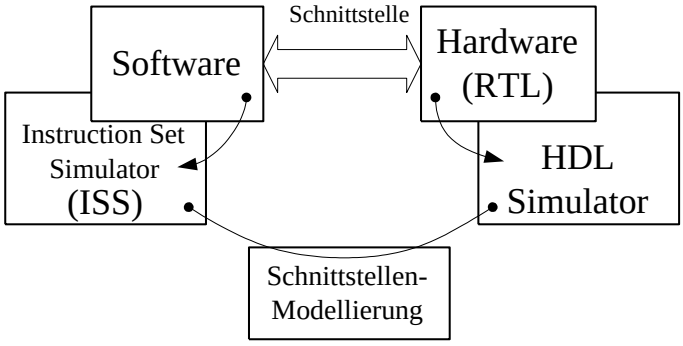

Abbildung 2.1: Hardware/Software-Coverifikation

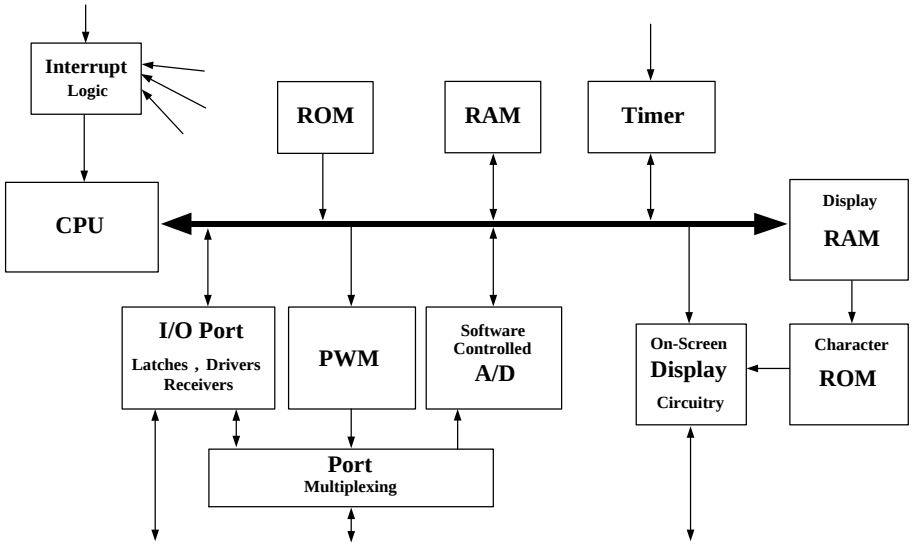

Abbildung 2.2: Typischer Aufbau eines Mikrocontrollers

Schaltungsbeschreibung in einer *Hardware Description Language* (kurz HDL) und der Softwareumgebung bringt auch praktische Implementierungsprobleme mit sich, da die Verifikationswerkzeuge von Hardware und Software, die intern traditionell grundsätzlich unterschiedlich aufgebaut sind, gekoppelt werden müssen [4, 5].

Ein weiterer Aspekt ist die steigende Integrationsdichte von integrierten Schaltungen, die einen Anstieg der Komplexität der zu entwickelnden Systeme zur Folge hat. Der Systementwurf muss daher auf einer höheren funktionellen Ebene beginnen [6]. Diese sogenannten *System-on-Chip* (kurz SoC)- Entwürfe beinhalten neben der Hardware für Schnittstellen und interne Logik auch Speicher und Prozessorkerne für die Ausführung von Software, wobei heutige Systeme einen Softwareanteil von 50 – 90 % haben [7]. Die Abbildung 2.2 zeigt ein typisches Beispiel für einen SoC-Entwurf.

Die gemeinsame Beschreibung von Hardware und Software verlangt gänzlich unterschiedliche Konzepte. Dem strikt sequentiellen Abarbeiten von Programmen steht die Parallelität von Hardwareeinheiten gegenüber. Eine gemeinsame Beschreibungssprache muss daher die Lücke zwischen Hardware und Software schließen. Eine hardwarenahe Sprache ist notwendig, damit eine hohe Simulationsperformance erreicht wird. Außerdem sollte die Wiederverwendung von bereits bestehendem Code verschiedener Hersteller (Intellectual Property, kurz IP) leicht möglich sein. Für die Entwicklung von Software muss die Sprache u.a. eine übersichtliche Beschreibung von Algorithmen sowie eine gute Strukturierbarkeit des Codes zulassen.

Die Sprache *SystemC*, die im Rahmen dieser Arbeit untersucht wird, verfolgt den Ansatz, C++ als Basis für den Hardware/Software-Entwurf zu verwenden, um den Anforderungen gerecht zu werden. C++ ist eine objektorientierte Programmiersprache, die

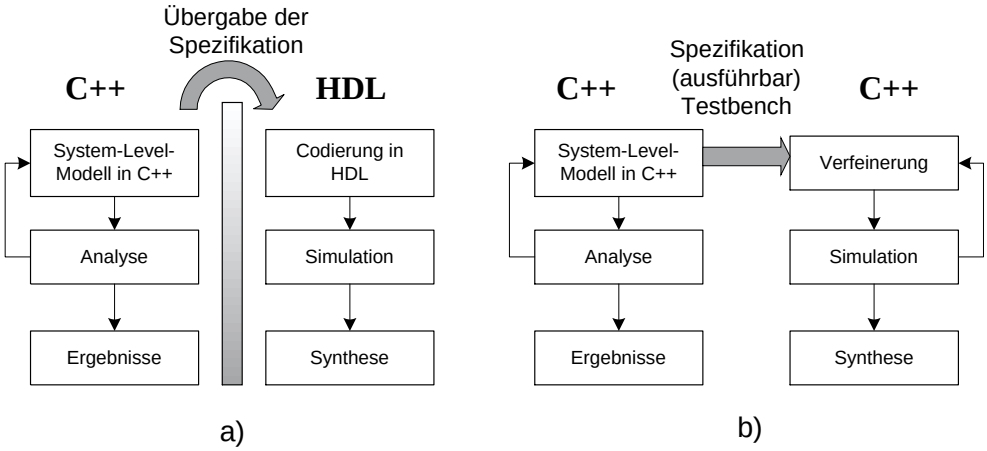

Abbildung 2.3: Vergleich zw. herkömmlichen und SystemC-Entwicklungsablauf

es ermöglicht, neue Datentypen und Methoden, die auf diesen operieren, zu definieren, ohne die Syntax erweitern zu müssen. Durch die Einführung einer neuen Klassenbibliothek, welche die notwendigen Konstrukte zur Modellierung von Zeitverhalten, Parallelität, synchronen und asynchronen Prozessen, Hierarchien und Hardware-Datentypen beinhaltet, wird die ursprünglich für Software entwickelte Sprache auch zur Beschreibung von Hardware erweitert.

C++ ist eine weit verbreitete Programmiersprache, mit der die meisten Softwareentwickler Erfahrung haben. Für die Hardwareentwickler reichen i.a. Kenntnisse der Untermenge C aus, über die viele von ihnen verfügen.

Eine Vielzahl von namhaften EDA[1]-Anbietern aus dem Bereich Systementwicklung[2] unterstützen SystemC mit der Vision, Interoperabilität zu erreichen und den Hardware/Software-Entwicklungsablauf zu verbessern. Die SystemC-Quelldateien sind deshalb auch für jedermann über das Internet zugänglich. Die Abbildung 2.3 zeigt eine Gegenüberstellung von HDL- und SystemC-Entwurfsablauf. Bei letzterem werden ausgehend von einer funktionalen Spezifikation des Gesamtsystems in C++ nach und nach Teile davon in Hardwarearchitekturen verfeinert. Die Verifikation kann dabei auf jeder Ebene im Entwurfsablauf einfach durch die Kompilierung der C++-Quelldateien und anschließende Ausführung erfolgen.

SystemC ist nicht der einzige Ansatz, die oben beschriebenen Probleme bei der gemeinsamen Entwicklung von Hard- und Software zu lösen. Daneben gibt es auch Sprachen wie Superlog, C++/Cynlib, SpecC, C-Level, Rosetta, CowareC, Java u.a.[8].

Im Rahmen der vorliegenden Arbeit soll untersucht werden, in wieweit sich SystemC als Grundlage für den Entwurf und die Verifikation eines typischen Hard-

[1] Electronic Design Automation

[2] Eine Auflistung findet man unter: *http://www.systemc.org/who_we_are.htm*

ware-Software-Systems eignet. Dieses besteht im Wesentlichen aus einem Ethernet-Controller und einem Baustein, der einen Prozessor und einen Speicher für die Ein- und Ausgabedaten beinhaltet. Konkret umfasst die Aufgabenstellung die Untersuchung folgender Punkte:

- Entwurf eines Systems auf funktionaler und Hardware-Ebene
- Abstraktionsgewinn bei der Beschreibung
- Performancegewinn bei der Simulation und die damit zusammenhängende Kostenreduktion
- Coverifikation mit Hardwarebeschreibungssprachen
- IP-Erstellung und -Nutzung
- Synthesefähigkeit
- Vorhandensein von EDA-Werkzeugen mit SystemC-Unterstützung überprüfen
- Durchgängiger Entwurfsablauf

Die Arbeit wurde in Kooperation mit Siemens AG Österreich (PSE ECT Technologiezentrum Systemverifikation) durchgeführt.

Im nächsten Kapitel erfolgt eine kurze Einführung in die Sprache SystemC mit Vergleichen zu reinen Hardwarebeschreibungssprachen. Dann wird anhand eines konkreten Systems der Entwurf, die Cosimulation von SystemC und VHDL sowie die Synthese untersucht. In der anschließenden Bewertung werden Antworten auf die Fragen der Aufgabenstellung geliefert.

Kapitel 3

Einführung in SystemC

Im September 1999 schlossen sich Firmen aus den Bereichen Halbleiter, IP, EDA und Embedded Software zur „Open SystemC Initiative“ (OSCI[1]) zusammen, zu deren Steering-Group u.a. ARM, Synopsys und Coware gehören. Ihr Ziel war es, den Austausch von IP-Modellen zu ermöglichen, sowie Codesign und Aufteilung von Hard- und Software zu unterstützen [4]. Das Produkt, genannt *SystemC*, ist ohne Lizenzgebühren erhältlich[2] und soll sich als De-facto-Standard für System-Level-Spezifikation durchsetzen. Da SystemC eine Klassenbibliothek ist, benötigt man zur Simulation lediglich einen ANSI C++ Compiler. SystemC ist deshalb plattformunabhängig.

3.1 Konzepte

Objektorientiertes Programmieren (kurz OOP) ist die Grundlage für moderne Softwareentwicklung. Unter den zahlreichen dafür verfügbaren Programmiersprachen ist C++ wahrscheinlich die am weitesten verbreitete, zumal es die Vorteile der Hardwarenähe, die es von seinem Vorgänger C geerbt hat, mit den problemorientierten Funktionen einer objektorientierten Sprache verbindet [9]. Die in C++ neu eingeführten Sprachfunktionen, die OOP unterstützen, sind insbesondere Klassen, Schablonen (*Templates*), *Overloading* von Funktionen und Operatoren sowie die Vererbung von Klasseneigenschaften.

Ein Programmierer kann eine Applikation in überschaubare Teile gliedern, indem neue Objekte definiert werden [10]. Diese mit Datenabstrahierung bezeichnete Technik entspricht dem Top-Down-Entwurf in der Hardwareentwicklung.

Im speziellen Anwendungsfall Hardwarebeschreibung können die Konzepte von OOP folgendermaßen genutzt werden:

[1] Synopsys *http://www.synopsys.com*
Coware *http://www.coware.com*
Frontier *http://www.frontierd.com* u.v.a.

[2] unter *http://www.systemc.org*

- Die Definition von Klassen führt zu abgeschlossenen Modulen, die einerseits als Behälter für Bauelemente zur Beschreibung des Aufbaus eines Systems dienen können und andererseits Bauelemente selbst beschreiben können.
- Durch Vererbung entstehen abgeleitete Klassen, deren gemeinsame Attribute in einer sogenannten Basisklasse zentral definiert sind. Grundlegende Eigenschaften wie z.B. die Bezeichnung und das angeschlossene Signal eines Ports könnten unabhängig von der Art des Ports (Eingang, Ausgang, ...) in einer Basisklasse deklariert werden.
- Templates erlauben die Definition von Objekten mit Parametern und bieten somit die Möglichkeit, Signale und Ports von unterschiedlichem Datentyp zu erzeugen.
- Durch Operator Overloading kann der Code lesbarer gestaltet werden, in dem z.B. Signalzuweisungen wirklich durch ein '='-Zeichen anstelle eines Funktionsaufrufes ausgedrückt werden.

Die genannten Konzepte wurden in einem Programm zur Simulation von Hardwarekomponenten in C++ angewendet. Dieser Simulator unterstützt Hierarchien, d.h. Module können eine beliebige Anzahl von Submodulen enthalten. Weiters ist man nicht auf den üblichen „boolean"-Signaltyp beschränkt, sondern man kann Signalen unterschiedlichste Datentypen zuweisen. Eine Ausgabe der Signalzustände dient zur Verifikation.

Listing 3.1 Beispiel einer Klassendefinition für einen Port

```
template<class T>
class out_port : public port {
   private:
      signal<T> *conn_sig;
      out_port<T> *conn_port;
   public:
      out_port();    // constructor
      void assign(signal<T> &);
      void assign(out_port<T> &);
      void write(T);
};
```

Das Listing 3.1 zeigt, wie die Klassendefinition für einen Ausgabe-Port namens `out_port` nach den oben beschriebenen Konzepten implementiert wurde. Diese Klasse ist von der Basisklasse `port` abgeleitet und wird mit einem Typenparameter `T` definiert. Die privaten Datenfelder `conn_sig` und `conn_port` sind Zeiger auf ein angeschlossenes Signal bzw. auf einen direkt verbundenen Port. Der von außen sichtbare Teil wird durch das Schlüsselwort `public` eingeleitet. Er umfasst den Constructor für die Initialisierung des Ports, `assign`-Methoden für die Zuweisungen von Ports und Signalen, sowie die Methode `write`, die einen Wert ausgeben soll. Letztere ist im Listing 3.2 abgedruckt. Da an einen Port ein Signal oder ein weiterer Port der oberen

Hierarchiestufe angeschlossen sein kann, erfolgt das Schreiben eines Wertes auf einen Port rekursiv bis ein Signal erkannt wurde, dem dann dieser Wert zugewiesen wird.

Listing 3.2 Write-Methode für einen Port

```
void out_port<bool>::write(bool v){
   if (conn_type == UNCON){
      cerr << "unconnected port" << endl;
      exit(-1);
   }
   if (conn_type == PORT)
      conn_port->write(v); // recursive port-write
   else
      conn_sig->assign(v); // until signal assigned
}
```

Ein Port vom Typ boolean kann innerhalb eines Moduls durch die Deklaration

`out_port<bool>` *Portname*;

instanziert werden.

Es wurden einfache Logikgatter (NAND, Inverter), ein RS-Latch bestehend aus diesen Gattern, sowie ein Addierer codiert und durch Simulation verifiziert. Die Testentwürfe konnten aus beliebig vielen Hierarchiestufen bestehen.

3.2 Syntax

Da die Syntax von SystemC jene von C++ ist, werden im Folgenden nur die Konstrukte erläutert, die in der SystemC-Bibliothek enthalten sind und durch Einbinden der Datei `systemc.h` verfügbar gemacht werden. Für eine ausführlichere Dokumentation der folgenden Zusammenfassung wird auf [11] verwiesen.

3.2.1 Module

Module dienen dazu, um komplexere Systeme in überschaubare Teile zu gliedern. Sie bilden Bausteine, deren interner Aufbau von außen nicht sichtbar ist und bieten dem Entwickler somit die Möglichkeit, die Funktionalität zu ändern, ohne die Schnittstelle zu beeinflussen. In einem Schaltplan würde ein Modul einem Bausteinsymbol entsprechen.

Module werden nach außen hin durch ihre Ports repräsentiert und können weitere Module, Signale, Prozesse und lokale Daten enthalten. Die Syntax lautet folgendermaßen:

`SC_MODULE` (*Modulname*) {

// Modulinhalt

};

bzw.

`struct` *Modulname* : `sc_module` {

// Modulinhalt

};

Das Makro SC_MODULE im ersten Fall soll lediglich die Lesbarkeit erhöhen. Beim Erzeugen einer Instanz eines Moduls wird der sogenannte *Constructor* aufgerufen, der die nötigen Initialisierungen für dieses Modul ausführt. Er wird durch das Makro

`SC_CTOR` (*Modulname*) {...}

angegeben. Eine sinnvolle Aufgabe eines Constructors könnte z.B. die Initialisierung des Speicherbereichs eines RAMs sein (siehe Listing 3.3)[3].

Listing 3.3 Initialisierung im Constructor

```
// ram.h
#include "systemc.h"
SC_MODULE(ram) {
   // Deklarationen
   int memdata[64];
   int i;
   SC_CTOR(ram){
      for (i=0; i++; i<64)
         memdata[i] = 0;
   }
};
```

3.2.2 Signale und Ports

Ports bilden die Schnittstelle eines Moduls nach außen. Die Verbindungen zwischen den Ports und damit den Modulen werden durch Signale hergestellt, die Prozesse in den Modulen aktivieren können. Es gibt drei Arten von Ports:

- Eingang: `sc_in`<*Porttyp*> *PortName*;
- Ausgang: `sc_out`<*Porttyp*> *PortName*;
- Bidirektional: `sc_inout`<*Porttyp*> *PortName*;

Ein Signal wird durch

`sc_signal`<*Signaltyp*> *SigName*;

definiert. Als Signal- und Porttyp kommen alle C++ built in - Typen (`int`, `char`, `double`, ...) sowie die SystemC-Datentypen (`sc_int<n>`, `sc_logic`, ...) in Frage. Somit können auch selbstdefinierte Strukturen und Objektklassen als Typ eingesetzt werden.

Beispiele:

```
sc_in<bool> x;          // Eingang vom Typ bool
```

[3] Die Deklaration der Signale, Ports und Funktionen soll hier noch nicht gezeigt werden. Entsprechende Beispiele vollständiger Module werden in den folgenden Kapiteln angeführt.

```
sc_out<int> y;          // C++ Integer Ausgang
sc_inout<sc_bit> q;     // Bidirektion. Port vom Typ sc_bit
sc_signal<sc_int<8> > dbus;   // 8 bit SystemC integer Signal
sc_in<sc_logic> a[32]; // Eingänge a[0] bis a[31] vom Typ sc_logic
sc_signal<bool> i[16]; // Signale i[0] bis i[15] vom Typ bool
```

Alle Prozesse innerhalb eines Simulationszyklus[4] operieren mit den alten Werten der Signale, wie man es von VHDL her gewohnt ist. Ein Port ist immer mit einem Signal oder einem Port der nächsthöheren Hierarchie verbunden, wobei die Signal- und Porttypen jeweils übereinstimmen müssen.

Wenn ein Signal mehr als einen Treiber besitzt, müssen Konflikte beim Zugriff aufgelöst (resolved) werden. SystemC stellt dazu eine vierwertige Logik mit den Werten '0', '1', 'X' (nicht bestimmbar) und 'Z' (hochohmiger Zustand) bereit, die durch die Definition von „Resolved Logic Vectors“ als Signale und Ports verwendet wird. Die Klassen dazu lauten:

- Resolved Logic Eingang: `sc_in_rv<`*n*`>` *PortName*;
- Resolved Logic Ausgang: `sc_out_rv<`*n*`>` *PortName*;
- Resolved Logic Bidirektional: `sc_inout_rv<`*n*`>` *PortName*;
- Resolved Logic Signal: `sc_signal_rv<`*n*`>` *SigName*;

Dabei bezeichnet *n* die Breite des Vektors.

Um den Fortschritt der Simulation zu steuern, benötigt man spezielle Signale – die Taktsignale. Durch Verwendung der Klasse `sc_clock` kann ein periodisches Signal erzeugt werden:

```
sc_clock clk_50m("clk_50m", 20, 0.5, 2, true);
```

Diese Definition erzeugt ein Takt-Objekt namens clk_50m, mit einer Periode von 20 Zeiteinheiten[5] und einem Tastverhältnis von 50%, deren erste Flanke nach 2 Zeiteinheiten auf einen Wert true führt. Es existieren für alle Parameter außer dem Namen Defaultwerte (Periode: 1, Tastverhältnis: 0.5, Zeitpunkt erste Flanke: 0, erster Wert: true).

3.2.3 Prozesse

Die Funktionalität von Modulen wird durch eine Summe von Prozessen gebildet. Im Constructor des Moduls wird festgelegt, welche der drei Prozessarten (Method-,

[4] ein englischer, aber gebräuchlicherer Ausdruck dafür ist Delta-Cycle

[5] Die Angabe von Zeiteinheiten statt richtigen Einheiten wie ns ist etwas verwirrend. Für die Simulation macht das aber keinen Unterschied. Zur Visualisierung in Waveform-Dateien wird als Einheit 1s angenommen, was bei manchen Betrachter-Programmen zu Problemen führt.

Thread-, Clocked Thread-Prozess) ausgeführt werden soll, welche Funktion dem Prozess zugeordnet ist und durch welche Signale der Prozess aktiviert wird. Die drei Prozesstypen werden im Folgenden beschrieben.

Method-Prozesse

Method-Prozesse werden aufgerufen, wenn sich ein Signal aus der Sensitivitätsliste ändert und übergeben nach ihrer Ausführung die Kontrolle wieder an den Simulator zurück. Durch

`SC_METHOD` (*Funktionsname*);

wird eine bestimmte Funktion, die zuvor im Modul deklariert werden muss, installiert. Die Sensitivitätsliste wird durch

`sensitive` << *Signal1* << *Signal2* ...

erzeugt. Verwendet man stattdessen `sensitive_pos` bzw. `sensitive_neg`, wird der Prozess nur durch die positive bzw. negative Flanke eines Signales aktiviert.

Listing 3.4 Method-Prozess: positiv flankengetriggertes D-Flip Flop

```
// dff_pos_edge.h
#include "systemc.h"

SC_MODULE(dff_pos_edge){
   sc_in<bool> clk;
   sc_in<bool> din;
   sc_out<bool> dout;
   void doit();
   SC_CTOR(dff_pos_edge){
      SC_METHOD(doit);
      sensitive_pos << clk;
   }
};

// dff_pos_edge.cpp
#include "systemc.h"
#include "dff_pos_edge.h"

void dff_pos_edge::doit(){
   dout = din;
}
```

Das Listing 3.4 zeigt als Beispiel für ein Modul mit einem Method-Prozess ein positiv flankengetriggertes D Flip Flop.

Thread-Prozesse

Im Gegensatz zu Method-Prozessen werden Thread-Prozesse nur einmal gestartet und durchlaufen immer wieder die gleiche Schleife, in der wait()-Kommandos zur vorübergehenden Unterbrechung dienen. Mit ihnen kann nahezu alles modelliert werden.

Der Arbitrierungsaufwand bei der Simulation ist jedoch aufwendiger als bei Method-Prozessen. Die Installation einer Funktion als Thread-Prozess erfolgt durch

SC_THREAD (*Funktionsname*);

mit anschließender Angabe der Sensitivitätsliste wie bei Method-Prozessen.

Listing 3.5 Thread-Prozess: Finite State Machine

```
// fsm.h
#include "systemc.h"

SC_MODULE(fsm){
   sc_in<bool> in1;
   sc_in<bool> in2;
   sc_out<bool> out1;
   sc_signal<int> state;
   void proc();
   SC_CTOR(fsm){
      SC_THREAD(proc);
      sensitive << in1 << in2;
   }
};

// fsm.cpp
#include "systemc.h"
#include "fsm.h"

void fsm::proc(){
   int state = 0;

   while (true){
      switch (state){
      case 0:
         // wait on in1
         while (in1.delayed() == false) wait();
         out1 = true;   // do something
         state = 1;     // set next state
         wait();
         break;
      case 1:
         // wait on in2
         while (in2.delayed() == false) wait();
          out1 = false; // do something
         state = 2;     // set next state
         wait();
         break;
      // other cases not shown
      default:
         // error message
         break;
      }
   }
}
```

Das Listing 3.5 zeigt einen Zustandsautomaten (Finite State Machine), der in Zustand 0 auf die Aktivierung von `in1` und in Zustand 1 auf die Aktivierung von `in2` wartet[6].

Clocked Thread-Prozesse

Clocked Thread-Prozesse sind synchrone Thread-Prozesse, deren Aktionen erst zur nächsten Taktflanke sichtbar werden. Im Unterschied zu den Thread-Prozessen erfolgt keine Angabe der Sensitivitätsliste sondern das zweite Argument im Aufruf

`SC_CTHREAD` (*Funktionsname, Taktflanke*);

spezifiziert, welche Flanke des Taktsignals (*Takt*.pos() für die positive bzw. *Takt*.neg() für die negative Taktflanke) den Prozess triggert. Mit Hilfe von Clocked Thread-Prozessen können implizite Zustandsautomaten (ohne explizite Angabe von Zuständen) in übersichtlicher Form realisiert werden, wie das Listing 3.6 zeigt.

Thread- und Clocked Thread-Prozesse laufen in Endlosschleifen. Oftmals ist es notwendig, Initialisierungen außerhalb einer Endlosschleife durchzuführen, wenn ein bestimmtes Ereignis eintritt (z.B. Reset). Um zu verhindern, dass bei jeder wait - Anweisung die gleiche Bedingung abgefragt werden muss, kann „*Watching*“ eingesetzt werden.

Bei *Globalem Watching* wird der Zustand eines Signals im gesamten Prozeß überprüft, in dem es angewendet wird. Globales Watching wird im Constructor des Modules durch die Anweisung

`watching` (*Signal*`.delayed() ==` *Wert*);

installiert. Sobald das *Signal* den somit definierten *Wert* erhält, wird an den Beginn der Prozessfunktion gesprungen, wo Anweisungen außerhalb der Endlosschleife stehen können.

Lokales Watching erlaubt eine genaue Angabe, welche Signale in welchem Bereich eines Prozesses überprüft werden. Durch die Angabe der vier Makros `W_BEGIN`, `W_DO`, `W_ESCAPE` und `W_END`) werden drei Bereiche abgegrenzt. Im Ersten befinden sich die Watching-Anweisungen, im Zweiten die Prozessfunktionalität und im Dritten die Event-Handler. Letztere sind jene Programmteile, die ausgeführt werden, wenn Watching-Signale aus dem ersten Bereich aktiv sind.

3.2.4 Hierarchien

Durch die Einführung von Hierarchien werden Entwürfe übersichtlicher und das Auffinden von Fehlern wird erleichtert. Das folgende einfache Beispiel zeigt, wie Hierarchien in SystemC aufgebaut werden. In der Abbildung 3.1 ist der Datenpfad eines digitalen Filters dargestellt, das aus den drei Submodulen *Sample*, *Coeff* und *Mult* besteht. Die

[6] Die Verwendung der delayed()-Funktion ist notwendig, um Übersetzungsfehler zu vermeiden.

Listing 3.6 Clocked Thread-Prozess: Multiplexer (Implizite State Machine)

```
// mux.h
#include "systemc.h"

SC_MODULE(mux_m){

   /* ports */
   sc_out<sc_uint<4> > data4_o;
   sc_in<sc_uint<32> > data32_i;
   sc_in<bool> en_i;
   sc_out<bool> en_o;
   sc_in_clk clk;

   SC_CTOR(mux_m){
      SC_CTHREAD(select, clk.pos());
   }
   void select();
};

// mux.cpp
#include "systemc.h"
#include "mux.h"

void mux_m::select(){
   sc_uint<32> data;

   while (true){
      while (en_i.delayed() == 0) wait();
      en_o = 1;
      data = data32_i.read();
      data4_o = data.range(3,0);
      wait();
      data4_o = data.range(7,4);
      wait();
      // and so on
      // ...
      data4_o = data.range(31,28);
      wait();
      en_o = 0;
      data4_o = 0;
   }
}
```

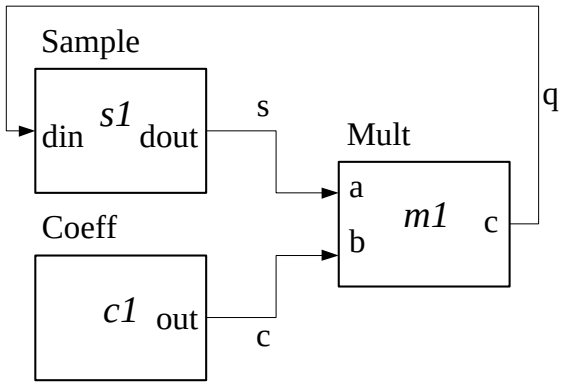

Abbildung 3.1: Verbindung von Modulen

Ports dieser Module sind durch die lokalen Signale *q*, *s* und *c* verbunden. Die äquivalente Beschreibung mit SystemC zeigt das Listing 3.7.

Listing 3.7 Verbindung von Modulen

```
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"

SC_MODULE(filter) {
   sample *s1;
   coeff *c1;
   mult *m1;
   sc_signal<sc_uint<32> > q, s, c;

   SC_CTOR(filter) {
      s1 = new sample ("s1");
      s1->din(q);
      s1->dout(s);
      c1 = new coeff ("c1");
      c1->out(c);
      m1 = new mult ("m1");
      m1->a(s);
      m1->b(c);
      m1->q(q);
   }
};
```

Die Moduldeklarationen der Submodule befinden sich idealer Weise in getrennten Dateien, die am Beginn des Hauptmoduls mittels „`#include`“-Direktiven eingebunden werden. Im Filter-Modul zeigen *s1*, *c1* und *m1* auf die entsprechenden Submodule. Diese anfangs nicht initialisierten Zeiger werden erst bei der Ausführung des Constructors durch das Schlüsselwort `new` auf die dynamisch erzeugten Submodule gelenkt. Nach der Instanzierung eines Moduls werden deren Ports mit Signalen der oberen Hier-

archie verbunden.

In diesem Beispiel werden die Namen der Ports bei der Verbindung explizit angegeben. Diese Art der Verbindung wird deshalb „*Named Connection*“ genannt. Die Ports vom Modul s1 könnten aber auch durch die Anweisung

```
(*s1)(q,s);
```

mit den Signalen verbunden werden. Diese mit „*Positional Connection*“ bezeichnete Methode ist zwar platzsparend, für größere Entwürfe aber nicht geeignet, da man die Übersicht rasch verlieren kann und die drohenden Verbindungsfehler schwer aufzuspüren sind.

3.2.5 Datentypen

SystemC stellt zusätzlich zu den C++-Datentypen speziell für hardwarenahe Modellierung noch folgende Datentypen zur Verfügung:

sc_bit	1 Bit für zweiwertige Logik
sc_logic	1 Bit für vierwertige Logik
sc_int	1 - 64 Bit vorzeichenbehaftete ganze Zahl
sc_uint	1 - 64 Bit vorzeichenlose ganze Zahl
sc_bigint	vorzeichenbehaftete ganze Zahl beliebiger Größe
sc_biguint	vorzeichenlose ganze Zahl beliebiger Größe
sc_bv	Bitvektor für zweiwertige Logik
sc_lv	Bitvektor für vierwertige Logik

sc_bit ist ein zweiwertiger Datentyp, der ein einzelnes Bit repräsentiert. Ein Objekt vom Typ sc_bit wird durch die Anweisung

`sc_bit` var;

deklariert und kann lediglich die Werte ’0’ und ’1’ annehmen.

Ein etwas allgemeinerer Datentyp ist `sc_logic`. Er kann die vier Werte ’0’, ’1’, ’X’ (unbestimmbar) und ’Z’ (hochohmig) annehmen. Mit ihm ist es möglich, Signale mit Reset-Verhalten, Tristate-Logik und mehreren Treibern zu modellieren. Er ist jedoch wesentlich aufwendiger bei der Simulation.

Viele Systeme benötigen arithmetische Operationen mit einer bestimmten Bitbreite. Diese werden durch die Datentypen *sc_int* und *sc_uint* bereitgestellt. Der vorzeichenbehaftete Typ sc_int wird im Zweierkomplement dargestellt. Objekte vom Typ sc_int bzw. sc_uint werden durch die Anweisungen

`sc_int`<*n*> var; bzw.

`sc_uint`<*n*> var;

deklariert, wobei *n* Werte zwischen 1 und 64 annehmen darf.

Für arithmetische Operationen mit einer Bitbreite größer als 64 wurden die Typen `sc_bigint` und `sc_biguint` eingeführt. sc_biguint steht für vorzeichenlose ganze Zahlen und sc_bigint für vorzeichenbehaftete ganze Zahlen in Zweierkomplement-Darstellung. Sie sind genauso zu verwenden wie sc_int bzw. sc_uint, mit dem Unterschied, dass sie keiner Begrenzung der Bitbreite auf 64 unterliegen. Der Einsatz dieser Datentypen sollte wirklich nur dort erfolgen, wo man mit den 64 Bit-Typen nicht mehr auskommt, weil sie für die Simulation wesentlich aufwendiger sind.

Die Datentypen *sc_bv* und *sc_lv* beschreiben Bitvektoren beliebiger Größe, wobei bei sc_bv eine zweiwertige Logik und bei sc_lv die selbe vierwertige Logik wie beim Typ sc_logic zugrunde liegt. sc_lv sollte wiederum nur dort verwendet werden, wo man mit zweiwertiger Logik nicht mehr auskommt. Objekte dieser Typen werden durch die Anweisungen

`sc_bv<`*n*`>` var; bzw.

`sc_lv<`*n*`>` var;

deklariert.

Eine genaue Auflistung der Operatoren für diese Datentypen findet man in [11].

3.2.6 Paket für Festkommazahlen

Beim Systementwurf auf abstrakter Ebene werden Algorithmen oft mit Gleitkommazahlen beschrieben. Um den Hardware-Aufwand für eine spätere Implementierung zu minimieren, ist es notwendig, Festkomma-Datentypen zu verwenden. SystemC stellt dazu vier Datentypen in einem eigenen Paket bereit:

- `sc_fixed`
- `sc_ufixed`
- `sc_fix`
- `sc_ufix`

Diese Typen sind speziell für digitale Signalprozessoren (DSP) eingeführt worden und für die meisten Anwendungen nicht relevant. Deshalb wird hier auch nicht näher darauf eingegangen.

3.2.7 System-Paket

Bei steigender Systemkomplexität ist es notwendig, die Modellierung auf einer höheren Abstraktionsebene vorzunehmen, und dieses funktionale Modell in weiteren Schritten zu verfeinern. Das System-Paket beinhaltet Erweiterungen, die diesen Top-Down-Entwurf unterstützen. Die Abbildung 3.2 zeigt den vorgeschlagenen Ablauf einer Sys-

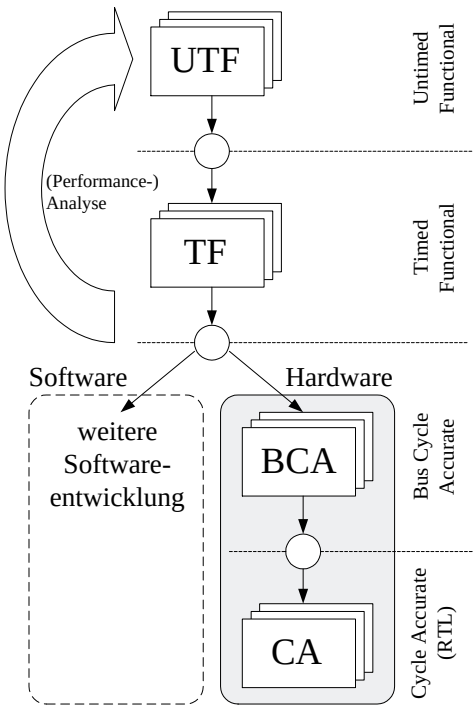

Abbildung 3.2: Entwurfsablauf in SystemC

tementwicklung. Zuerst wird ein Untimed Functional (kurz UTF) Modell erstellt, welches in C++ geschrieben ist und Algorithmen sequentiell abarbeitet. Danach erfolgt eine Aufteilung in zusammengehörige Blöcke (Master- und Slave- Prozesse in SystemC), die über abstrakte Ports kommunizieren. Durch die Zuweisung einer Ausführungszeit zu einem Prozess erhält man ein Timed Functional (kurz TF) Modell, mit dem die Performance der Algorithmen untersucht werden kann[7]. Nach der Partitionierung in Hard- und Software erfolgt erstmals eine Festlegung der Architektur. In dem sogenannten Bus Cycle Accurate (kurz BCA) Modell wird die Schnittstelle zu anderen Blöcken inklusive Timing festgelegt, während die interne Funktion weiterhin als zeitlos betrachtet wird. Werden schließlich alle Prozesse zeitabhängig modelliert, erhält man das endgültige, synthetisierbare Cycle Accurate (kurz CA) Modell auf der Hardwareseite. Der Softwareflow ist noch im Entwicklungsstadium. Die Überführung der Schaltungsbeschreibung muss manuell erfolgen.

Die Modellierung auf dem UTF-Level wird durch *Remote Procedure Calls* (kurz RPC) durchgeführt. Das ist ein Master-Slave Prinzip, bei dem ein aufrufender Prozess (Master) über einen speziellen Port einen Slave-Prozess aktiviert und erst nach dessen Beendigung mit seiner Aufgabe fortfährt. Der Master-Prozess kann ein beliebiger

[7] noch nicht enthalten in Version 1.1.

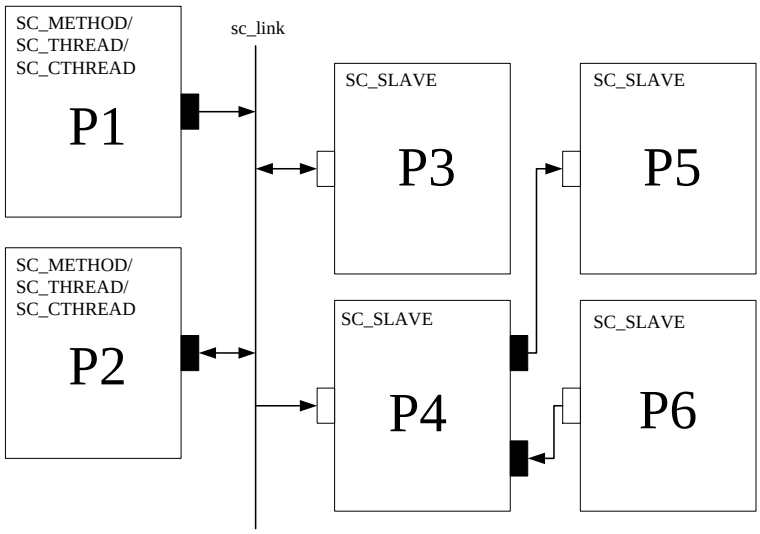

Abbildung 3.3: Beispiel für eine Master-Slave Konfiguration

Prozess sein (Method-, Thread- oder Clocked Thread-Prozess), der Slave-Prozess wird durch

`SC_SLAVE` (*Funktionsname*, *Slave-Port*);

im Constructor definiert. Ein Prozess kann mehrere Master-Ports haben, jedoch nur einen Slave-Port. In der Abbildung 3.3 ist eine mögliche Konfiguration von Mastern und kaskadierten Slaves dargestellt. Lediglich autonome Prozesse, also Prozesse ohne Slave-Port, wie die Prozesse P1 und P2 im Bild, laufen parallel ab. Slave-Ketten wie z.B. P4-P5 oder P4-P6 werden sequentiell abgearbeitet. Diese Mischung von Parallelität und Serialität erlaubt die gemeinsame Modellierung von Strukturen (Hardware) und Algorithmen (Software) im UTF-Level.

Master und Slaves kommunizieren über abstrakte Ports, die durch die Ausdrücke

- Master: `sc_master`<*Porttyp*> *PortName*;
- Master-Eingang: `sc_inmaster`<*Porttyp*> *PortName*;
- Master-Ausgang: `sc_outmaster`<*Porttyp*> *PortName*;
- Master-Bidirektional: `sc_inoutmaster`<*Porttyp*> *PortName*;
- Master: `sc_slave`<*Porttyp*> *PortName*;
- Slave-Eingang: `sc_inslave`<*Porttyp*> *PortName*;
- Slave-Ausgang: `sc_outslave`<*Porttyp*> *PortName*;
- Slave-Bidirektional: `sc_inoutslave`<*Porttyp*> *PortName*;

ähnlich wie normale Ports definiert werden. Die Verbindung zwischen den Ports wird durch

`sc_link_mp`<*Signaltyp*> *SigName*;

hergestellt, wobei die Zuordnung zu den Ports wie bei Signalen funktioniert. Wie die Endung „mp“ andeutet, handelt es sich dabei um Multipoint-Links, d.h. es können mehrere Master und Slaves miteinander verbunden werden. Wenn ein Master auf einen Port schreibt, werden alle Slaves, deren Slave-Port ein Eingang oder Bidirektional ist aktiviert. Wenn ein Master von einem Port liest, werden alle Slaves, deren Slave-Port ein Ausgang oder Bidirektional ist, aktiviert. Es muss jedoch sichergestellt sein, dass nur ein Slave auf den Port schreibt. Bei sc_inoutslave-Ports kann die Methode `input()` verwendet werden, um die Richtung des Transfers zu bestimmen. Sie liefert wahr, wenn der Port als Eingang verwendet wird.

Master- und Slave-Prozesse können Datentransfers mit einem Index über sogenannte *Indexed Ports* durchführen. Damit ist es z.B. möglich, einen Speicherbereich über Modulgrenzen hinaus mit einer Adresse anzusprechen. Indexed Ports werden durch Angabe eines zweiten Parameters in der Portdefinition erzeugt:

`sc_outmaster`<*Porttyp*, `sc_indexed`<*Bereich*> > *SigName*;

`sc_inslave`<*Porttyp*, `sc_indexed`<*Bereich*> > *SigName*;

Mir der Methode `get_address()` kann der Slave die vom Master angelegte Adresse ermitteln. Beispiele dafür findet man in Kapitel 4.2.

Auch eine Kombination von verschiedenen Abstraktionsebenen (UTF und CA) ist möglich. Genaue Details würden hier zu weit führen. Außerdem ist die Sprachdefinition des System-Pakets erst wenige Monate alt und speziell in diesem Bereich noch nicht vollständig.

3.2.8 Simulation

SystemC verfügt über einen zyklusbasierten Simulationskernel, d.h. die Signale werden bei den Taktflanken aktualisiert. Alle Prozesse, deren Eingangssignale sich verändert haben, werden ausgeführt, und erst dann werden ihre Ausgangssignale aktualisiert.

Der Simulationskernel durchläuft folgende Schritte:

1. Aktualisierung aller Taktsignale, deren Wert sich zum aktuellen Zeitpunkt ändert
2. Ausführung der Method- und Thread-Prozesse, deren Eingänge sich geändert haben, wobei Thread-Prozesse nur bis zum nächsten *wait()* ausgeführt werden
3. Getriggerte Clocked Thread-Processe werden in einer Warteschlange gespeichert, um im Schritt 5 ausgeführt zu werden; alle Signale und Ports werden aktualisiert
4. Die Schritte 2 and 3 werden wiederholt, bis alle Signale stabil sind

5. Ausführung der Clocked Thread-Prozesse aus der Warteschlange

6. Erhöhung der Simulationszeit und Sprung zu Schritt 1, wenn das Ende der Simulation noch nicht erreicht ist

Die Simulation muss von der obersten Ebene des Entwurfs, in der Funktion `sc_main()`, gestartet werden. Die Funktion `main()` wird vom Simulator gebraucht, und kann deshalb nicht verwendet werden. Ihre Argumente werden an `sc_main()` übergeben. Durch den Aufruf

`sc_start`(*Simulationsdauer*);

wird die Simulation gestartet. Ist das Argument negativ, läuft die Simulation unendlich lange. Der Befehl `sc_stop()` kann global verwendet werden und erzwingt die Beendigung der Simulation. Die aktuelle Zeit erhält man mit der Funktion `sc_time_stamp()`. Eine alternative Möglichkeit zur Simulationskontrolle besteht durch die Verwendung der Befehle `sc_initialize()` und `sc_cycle()`. Das Listing 3.8 zeigt die Funktion `sc_main()` mit den zuletzt genannten Befehlen.

Die Ausgabe von *Waveforms* wird in den drei Formaten

- VCD (Value Change Dump),
- ASCII WIF (Waveform Intermediate Format), und
- ISDB (Integrated Signal Data Base)

unterstützt. Da die Angabe der aufzuzeichnenden Signale und Variablen nur in `sc_main()` erfolgen darf, können nur Signale und Variablen von Modulen, jedoch keine lokalen Variablen von Funktionen protokolliert werden.

Eine genaue Beschreibung der zugehörigen Funktionen ist aus [11] zu entnehmen.

3.3 Gegenüberstellung SystemC – VHDL

Um die Unterschiede und Gemeinsamkeiten von SystemC und VHDL aufzuzeigen, werden an dieser Stelle zwei Beispiele behandelt.

3.3.1 Beispiel 1: D Flip Flop

Das Listing 3.9 zeigt den typischen Code eines D Flip Flop mit asynchronem Resetsignal in VHDL und das Listing 3.10 die entsprechende Implementierung in SystemC.

Die Abspaltung der Definitionen des SystemC-Codes in ein eigenes Header-File trägt wesentlich zur Übersichtlichkeit bei und weist Ähnlichkeiten mit dem Entity-Architecture-Konzept von VHDL auf. Einen wesentlichen Unterschied stellt der Constructor dar, der Funktionen zu Prozessen zuordnet und deren Sensitivität auf Signale definiert.

Listing 3.8 Beispielhafte sc_main-Funktion

```
#include "systemc.h"
// other header-files

int sc_main(int argc, char *argv[]){

   sc_clock clk50("clk50");

   // signals for connecting modules
   sc_signal<bool> s1;
   ...

   // modules
   register reg1("reg1");
   reg1.clk(clk50);
   reg1.en(s1);
   ...

   // create VCD file
   sc_trace_file *tf = sc_create_vcd_trace_file("trace");
   // specify traces
   sc_trace(tf, clk50.signal(), "clk50");
   sc_trace(tf, s1, "s1");
   ...

   // initialize signals
   s1 = false;
   ...

   // initialize simulation kernel
   sc_initialize();

   // simulate 1000 cycles
   for (long i=0; i<1000; i++){
      clk50 = 1;
      sc_cycle(10);
      clk50 = 0;
      sc_cycle(10);
   }
   return 0;
}
```

Listing 3.9 D Flip Flop mit asynchronem Reset (VHDL)

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

entity dff is
   port(
      clock : in std_logic;
      reset : in std_logic;
      din : in std_logic;
      dout : out std_logic);
end dff;

architecture rtl of dff is
begin
   process(reset, clock)
      begin
      if reset = '1' then dout <= '0'
      elsif clock'event and clock = '1' then
         dout <= din;
      end if;
   end process;
end rtl;
```

Listing 3.10 D Flip Flop mit asynchronem Reset (SystemC)

```
// dff.h
#include "systemc.h"

SC_MODULE(dff) {
   sc_in<bool> din, clock, reset;
   sc_out<bool> dout;

   void doit();

   SC_CTOR(dff) {
      SC_METHOD(doit);
      sensitive_pos << clock;
      sensitive << reset;
   }
};

// dff.cc
#include "systemc.h"
#include "dff.h"

void dff::doit() {
   if (reset) dout = false;
   else if (clock.event()) dout = din;
}
```

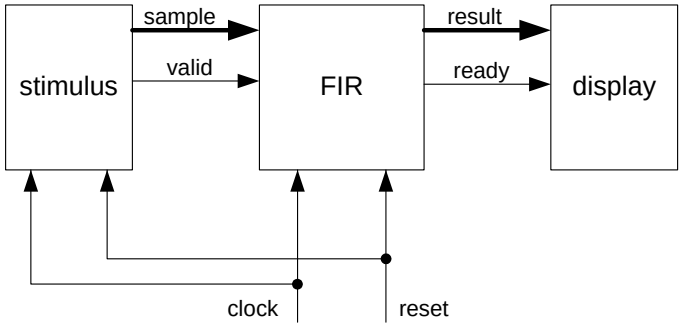

Abbildung 3.4: Testbench des FIR-Filters

3.3.2 Beispiel 2: FIR-Filter

Das FIR-Filter ist ein digitales Filter, das eine gewichtete Aufintegration der Eingangsdaten nach folgender Formel durchführt:

$$p(t) = \sum_{1 \leq n \leq N} h(nT) \cdot s(t - nT) \tag{3.1}$$

Die VHDL-Implementierung des Filters ist eine synthetisierbare RTL-Beschreibung, die die Komponenten FSM, ALU, RAM und ROM (für den Koeffizientenvektor h) enthält. Die SystemC-Implementierung vereint alle diese Blöcke in einem einzigen Block namens FIR, wie in der Abbildung 3.4 dargestellt ist.

Der Ausschnitt aus dem FIR-Modul der SystemC-Beschreibung, der die eigentliche Funktion enthält, ist im Listing 3.11 dargestellt. Er vereint im Gegensatz zum VHDL-Code die Kontroll- und Datenlogik in einem Modul und beschreibt die Aufsummation von Produkttermen nach Gleichung 3.1 in übersichtlicher, algorithmischer Form.

Die Beschreibung erfolgte durch 1400 Zeilen in VHDL und durch 180 Zeilen in SystemC, wobei in VHDL die fertigen Speicherblöcke einen Großteil des Codes ausmachen. Die Simulation von 20 ms dauerte auf einer Ultra-30 Workstation bei VHDL (Modelsim) 50 s und erzeugte eine 6,2 MB Trace- (.wav) Datei. Selbstgeschriebene, einfache Speicher ohne Timing reduzieren die Anzahl der Codezeilen auf 650 und verkürzen die Simulation auf 30 s. Die Ausführung des kompilierten SystemC-Programmes dauerte hingegen 27 s, trotzdem ein Tracefile (.vcd) der Größe 22 MB erzeugt wurde, was keine wesentliche Steigerung der Simulationsperformance bedeutet. Es konnte ein Faktor drei gewonnen werden, indem das Taktsignal nicht aufgezeichnet wurde und das Tracefile auf 1,5 MB zusammenschrumpfte. Man sollte also das Aufzeichnen von häufig wechselnden Signalen – wenn nicht unbedingt erforderlich – vermeiden, weil das '.vcd'-Traceformat sehr speicher- und zeitaufwendig ist. Die Ergebnisse sind in der nachfolgenden Tabelle zusammengefasst.

	VHDL		SystemC	
	mit Timing	ohne Timing	mit Trace	ohne Trace
Codezeilen	1400	650	180	180
Sim.dauer für 20 ms	50 s	30 s	27 s	9 s
Tracefilegröße	6,2 MB	6,2 MB	22 MB	1,5 MB

Listing 3.11 FIR Codeausschnitt

```
while (true){
   // einlesen
   wait_until(input_data_ready.delayed() == true);
   ram[offset] = sample.read();
   acc = 0;
   wait();
   // Operation
   for (int i = 0; i < 16; i++){
      acc += ram[(offset+i)&15]* coeffs[i];
      wait();
   }
   // schreiben
   result = acc;
   output_data_ready = 1;
   offset--;
   wait();
   output_data_ready = 0;
}
```

Aussagen von OSCI-Mitgliedern, wonach die SystemC-Simulation 50- bis 100-fach schneller als herkömmliche Verilog- und VHDL- Simulationen sind, können zumindest anhand des FIR-Beispiels nicht verifiziert werden. Hier konnte lediglich ein Faktor 3 erreicht werden. Es wäre auch kaum glaubwürdig, dass C++-Compiler wie gcc einen in dem Maß effektiveren Maschinencode als HDL-Compiler (vcom) bei annähernd gleicher Komplexität der Beschreibung erzeugen.

Ein kritischer Punkt bei solchen Vergleichen ist der Abstraktionslevel des Designs. Abstraktere Beschreibungen haben natürlich auch eine schnellere Simulation zur Folge. Der Vorteil von SystemC liegt weniger in der verkürzten Simulationsdauer, sondern es unterstützt die Beschreibung auf abstrakterer Ebene, was sich schlussendlich positiv auf die Simulationsperformance auswirkt. Es hat macht daher wenig Sinn, ein D Flip Flop oder ein NAND-Gatter mit SystemC zu beschreiben [12]. Bei einem Vergleich am Beispiel eines D Flip Flops dauerte die Ausführung des SystemC-Programmes sogar länger als die entsprechende VHDL-Simulation mit Modelsim.

Einen großen Einfluss auf die Performance der Simulation hat neben der schon erwähnten Größe des Tracefiles die Auswahl der Datentypen. Um deren Auswirkungen zu untersuchen, wurde der Quellcode des FIR-Filter-Beispiels zur Gänze von SystemC-

eigenen Datentypen befreit, d.h. alle sc_int- und sc_uint-Typen wurden durch den C++-Typ int ersetzt. Die Simulation zeigte jedoch keine spürbare Performancesteigerung. Dafür kann es zweierlei Gründe geben. Erstens, die ganzzahligen SystemC-Datentypen wurden sehr effizient implementiert. Und zweitens, die arithmetischen Befehle im FIR-Filter nehmen im Vergleich zum Overhead der Simulation und der restlichen Schaltungsbeschreibung einen verschwindenden Teil ein.

Um den letzteren Einfluss auszuschalten, wurde ein Programm geschrieben, dass aus einer einzigen Schleife bestand, in der arithmetische und logische Befehle ohne Simulationsoverhead ausgeführt wurden. Die Ergebnisse sind in folgender Auflistung zusammengefasst:

	`int`	`sc_int<32>`	`sc_bigint<32>`
Addition (+)	1.0	$\times$ 2.67	$\times$ 25.0
Multiplikation (*)	1.0	$\times$ 2.67	$\times$ 32.0
Schiebeoperation (<<)	1.0	$\times$ 2.25	$\times$ 32.0
Logisches Und (&)	1.0	$\times$ 2.25	$\times$ 28.0

Die Zahlenwerte sind immer relativ zu jenen vom Typ int dargestellt. Man erkennt, dass der Datentyp sc_int nur etwas mehr als die doppelte Simulationsdauer benötigt, was auf eine sehr effiziente Implementierung dieser Klasse schließen lässt. Bei der Verwendung einer Variable vom Typ sc_bigint mit gleicher Bitbreite explodiert die Simulationsdauer im günstigsten Fall der Addition auf den 25-fachen Wert. Dabei ist die Bitbreite 32 noch nicht einmal ein typischer Wert für diesen Datentyp, der erst ab 65 Bit zum Einsatz kommen sollte.

Die Wahl des richtigen Datentyps ist wichtig, sollte aber nach einer Abwägung der gewonnenen Ergebnisse nicht überbewertet werden. Sie kommt auf abstrakter Ebene eher zum tragen, weil hier der Simulations- und Kommunikationsoverhead viel geringer ist. Da aber die Simulationsperformance bei funktionellen Beschreibungen in der Regel ein zweitrangiges Problem ist, wird der Einfluss des Datentyps als nicht so entscheidend erachtet.

Kapitel 4

Anwendungsbeispiel: Ethernet-Controller

In diesem Abschnitt werden die Systementwurfs- und Verifikationseigenschaften von SystemC anhand eines konkreten praxisnahen Systems untersucht, dessen Herzstück ein Ethernet-Controller ist.

4.1 Modellierungsvorgaben

Das Gesamtsystem ist Teil eines digitalen Sprachvermittlungssystems und besteht aus mehreren Baugruppen mit einer Vielzahl von ASIC's. Die Arbeit konzentriert sich auf den in der Abbildung 4.1 gezeigten Ausschnitt. Die Modellierung umfasst das Interface IBUS-IF in ASIC1, welches Stimuli-Daten über den IBUS an den ATM-Controllerbaustein ATMC überträgt, der diese Daten in seinen Speicher schreibt. Der IO-Controller greift über die i486-ähnliche Schnittstelle auf den Speicher des ATMC zu und übermittelt die erhaltenen Daten über ein Media Independent Interface (kurz MII) an einen Physical Media Dependent (kurz PMD) - Baustein, der die eigentliche Umsetzung auf Ethernet erledigt, die in der Modellierung aber nicht mehr berücksichtigt wird. Der Datenfluss erfolgt in beide Richtungen.

Am IBUS werden die Daten bitseriell mit einer Taktrate von 50 Mbit/s mittels eines Software-Handshake-Verfahrens übertragen. Das i486-Interface ist mit seinem 32 Bit Daten- und Adressbus dem Intel486-Bus sehr ähnlich. Das MII entspricht dem Standard IEEE802.3u mit jeweils 4 Bit parallelen Ein- und Ausgängen, synchron zu einem 25 MHz Eingangstakt, mit Enable und einem Signal für die Kollisionserkennung. Eine genauere Beschreibung der Schnittstellen erfolgt in Kapitel 4.3.

Die Modelle für die Schnittstellen IBUS und MII sowie der ATMC bilden zusammen die Testbench für den IO-Controller und sollen auf Systemebene – ohne Details – implementiert werden. Für den IO-Controller wird ein vereinfachtes Modell angenom-

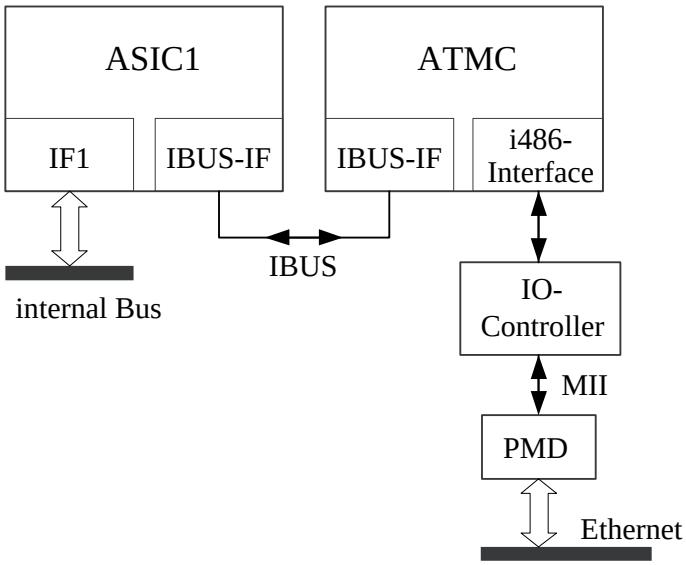

Abbildung 4.1: Ausschnitt aus dem Gesamtsystem

men, welches sukzessiv verfeinert werden und am Ende voll synthetisierbar sein soll.

Die Software im ATMC kommuniziert mit dem IO-Controller über Speicherbereiche, deren Aufbau in den Abbildungen 4.2 und 4.3 dargestellt ist. Die Frame-Buffer sind Speicherbereiche, die Datenrahmen von 400 Doppelwörtern (1600 Bytes) aufnehmen. Der Zeiger *TxFramePtr* verweist auf einen zu sendenden Frame. *RxPtrArray* ist ein Vektor (64 oder 128 Einträge), der von der Software mit Zeigern auf freie Frame-Buffer gefüllt wird. TxFramePtr und RxPtrArray befinden sich an festgelegten Adressen, die dem Controller und der Software bekannt sind.

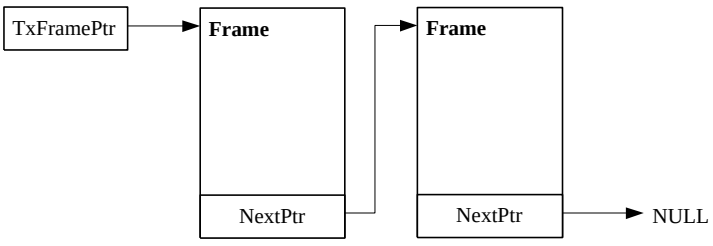

Abbildung 4.2: Speicherorganisation für die Ausgabe

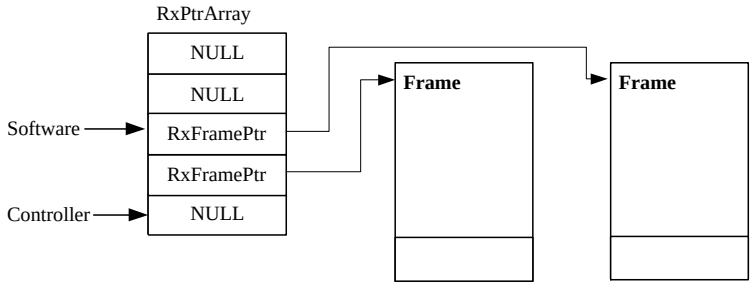

Abbildung 4.3: Speicherorganisation für die Eingabe

4.2 Beschreibung auf abstrakter Ebene

Um den angestrebten Top-Down-Entwurf zu verfolgen, wurde das gesamte System mit Hilfe des System-Pakets, dessen Grundzüge in Kapitel 3.2.7 erklärt sind, vorerst auf abstrakter Ebene definiert. In der Abbildung 4.4 sind die vier Module Stimuli IBUS, ATMC, IO-Controller und Stimuli MII sowie deren Verbindung untereinander dargestellt. Die Kommunikation zwischen den Modulen erfolgt ausschließlich über abstrakte Ports. An den Modulgrenzen bedeuten schwarz ausgefüllte Rechtecke Master-Ports und unausgefüllte Rechtecke Slave-Ports.

Die Stimuli-Module lesen die zu sendenden Daten von Quelldateien (ibus.stim, mii.stim) und schreiben erhaltene Daten auf Senken (ibus.out, mii.out), wodurch die Verifizierung erheblich erleichtert wird[1]. Außerdem erspart das Einlesen aus Stimulidateien das oftmalige Neukompilieren des Entwurfs. Eine Loggdatei, in der bestimmte Zustände mit einem Zeitstempel protokolliert werden, ist speziell bei Prozessen mit komplizierterer Zustandsabfolge eine fast unerlässliche Hilfe beim Auffinden von Fehlern. Die Ausgabe auf diese Datei kann durch einen Schalter im Makefile von der Kompilierung ausgeschlossen werden, um die Simulationsperformance zu erhöhen.

Der Read-Prozess des ATMC wird bei jedem ankommenden IBUS-Datum aktiviert und füllt diese in einen Speicher, der wie in der Abbildung 4.2 als verkettete Liste organisiert ist, mit dem Unterschied, dass am Ende eines Frames ein Feld namens *word_cnt* hinzukommt. Das ist notwendig, weil irgendwo markiert werden muss, welche Frames bereits versendet wurden bzw. welche Frames fertig zum Auslesen sind. Es stellt also einen Kommunikationsmechanismus zwischen ATMC (Software) und IO-Controller (Hardware) dar. Im Write-Prozess wird das Feld rx_array der Abbildung 4.3 linear nach

[1] Die Stimuli-Befehle werden in Kapitel 4.5 beschrieben.

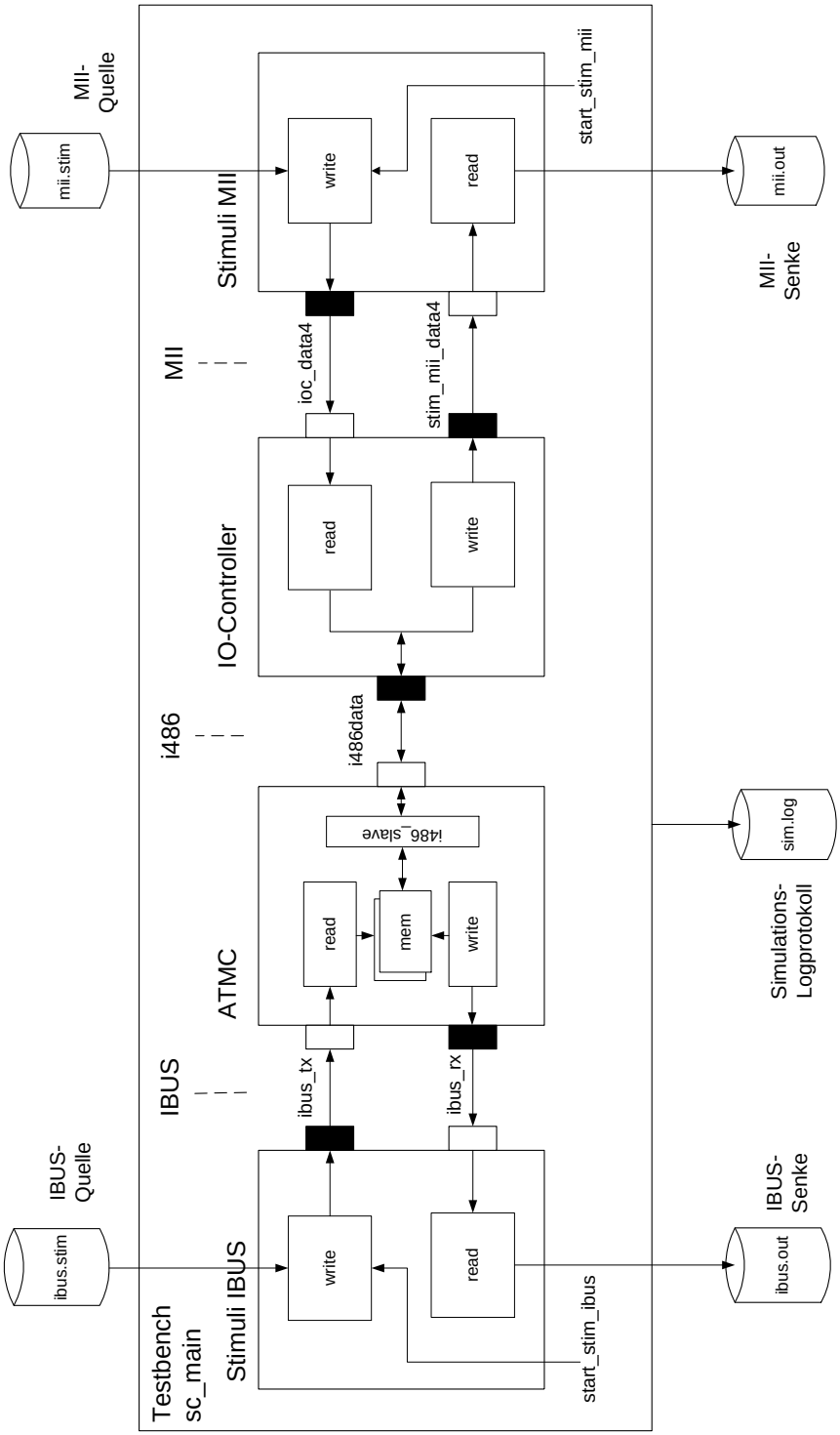

Abbildung 4.4: Gesamtsystem auf abstrakter Ebene

Zeigern auf gültige Adressen ($\neq 0$) durchsucht. Wenn ein entsprechender Zeiger gefunden wird, wird word_cnt des Frames, der damit referenziert wird, auf einen Wert ungleich Null überprüft. Trifft dies zu, wird der komplette Frame byteweise auf den IBUS geschrieben.

Listing 4.1 Modulaufbau des ATMC

```
SC_MODULE(mbdatm_m){
   /* ports */
   sc_outmaster<unsigned char> ibus_tx_o;
   sc_inslave<unsigned char> ibus_rx_i;
   sc_inoutslave<unsigned long,
      sc_indexed<MEM_SIZE> > i486data_io;
   sc_in_clk clk;
   /* variables */
   mii_frame *tx_frame_ptr;
   mii_array rx_array;

   SC_CTOR(mbdatm_m){
      SC_SLAVE(read_ibus, ibus_rx_i);
      SC_METHOD(write_ibus);
      sensitive_pos << clk;
      SC_SLAVE(i486_slave_proc, i486data_io);
      /* Initialize */
      tx_frame_ptr = NULL;
   }
   // ... function declaration
};
```

Im IO-Controller existieren zwei Prozesse. Der Write-Prozess ist für das Versenden in Richtung MII zuständig und erkennt sendebereite Frames daran, dass im Speicher des ATMC tx_frame_ptr ungleich Null und word_cnt gleich der Framegröße (400) ist. In diesem Fall wird der komplette Frame über das i486-Interface gelesen und in 3200 Schritten 4 Bit-weise auf das MII geschrieben. Damit vom ATMC erkannt wird, dass der Frame abgearbeitet wurde, wird noch 0xFFFFFFFF in das Feld word_cnt geschrieben. Dann wird der Zeiger auf den nächsten Frame gelesen und die Schleife beginnt wieder von vorne. Der Read-Prozess sammelt Nibbles vom MII und setzt sie zu 32 Bit-Doppelwörtern zusammen. Sobald alle 400 Doppelwörter eingelangt sind, wird ein rx_frame_ptr aus dem ATMC gelesen, der die Startadresse darstellt, auf die die eingelangten Daten über das i486-Interface geschrieben werden. Anschließend wird word_cnt auf 0xFFFFFFFF gesetzt, damit der ATMC einen vollen Frame erkennt und verarbeiten kann.

Die Moduldeklarationen des ATMC und des IO-Controllers sind in den Listings 4.1 und 4.2 abgedruckt. Sie zeigen, wie abstrakte Ports und Slave-Prozesse definiert werden. Die Schnittstelle zwischen den beiden Modulen wird durch abstrakte, indizierte (indexed) Ports gebildet. Sie sind prädestiniert dafür, modulinterne Speicherstellen von außen zu adressieren und daher besonders gut geeignet, die i486-ähnliche Schnittstelle

Listing 4.2 Modulaufbau des IO-Controllers

```
SC_MODULE(io_controller_m){
   /* ports */
   sc_inoutmaster<unsigned long,
      sc_indexed<MEM_SIZE> > i486data_io;
   sc_outmaster<sc_uint<4> > mii_data4_o;
   sc_inslave<sc_uint<4> > mii_data4_i;
   sc_in_clk clk;
   /* variables */
   sc_uint<32> in_fifo[MII_FRAME_SIZE];
   sc_uint<32> out_fifo[MII_FRAME_SIZE];
   unsigned long addr_tx_frame_ptr;
   unsigned long rx_ptr_array;

   SC_CTOR(io_controller_m){
      SC_SLAVE(control_read, mii_data4_i);
      SC_METHOD(control_write);
      sensitive_pos << clk;
      /* Initialize */
      for (int i = 0; i <  MII_FRAME_SIZE; i++)
         in_fifo[i] = out_fifo[i] = 0;
   }
   void control_write();
   void control_read();
};
```

zu beschreiben.

Bei jedem Schreib- oder Lesezugriff des IO-Controllers auf das indizierte Port i486data_io wird der i486_slave-Prozess im ATMC aktiviert, dessen Quellcode im Listing 4.3 dargestellt ist. Dort wird zuerst die Adresse des Zugriffs ermittelt. Je nachdem, ob ein lesender oder schreibender Zugriff erfolgt ist, wird dann entweder der Inhalt der adressierten Speicherstelle an das Port gelegt oder die Daten des Ports an der richtigen Stelle abgespeichert. Der Quellcode dieses Prozesses ist sehr kompakt und beschreibt in übersichtlicher Form die Funktionalität der i486-Schnittstelle.

Listing 4.3 Der i486_slave-Prozess

```
void mbdatm_m::i486_slave_proc(){

   mem_ptr address =
      (mem_ptr) i486data_io.get_address();
   if (i486data_io.input())
      *address = i486data_io.read();
   else
      i486data_io = *address;
}
```

Die Performance der Simulation dieses UTF-Modells wurde mit jener des verfeinerten CA-Modells, das im Kapitel 4.4 beschrieben ist, verglichen. Als Ergebnis sind die Simulationsdauern in Abhängigkeit der Anzahl der injizierten IBUS- bzw. MII-Frames

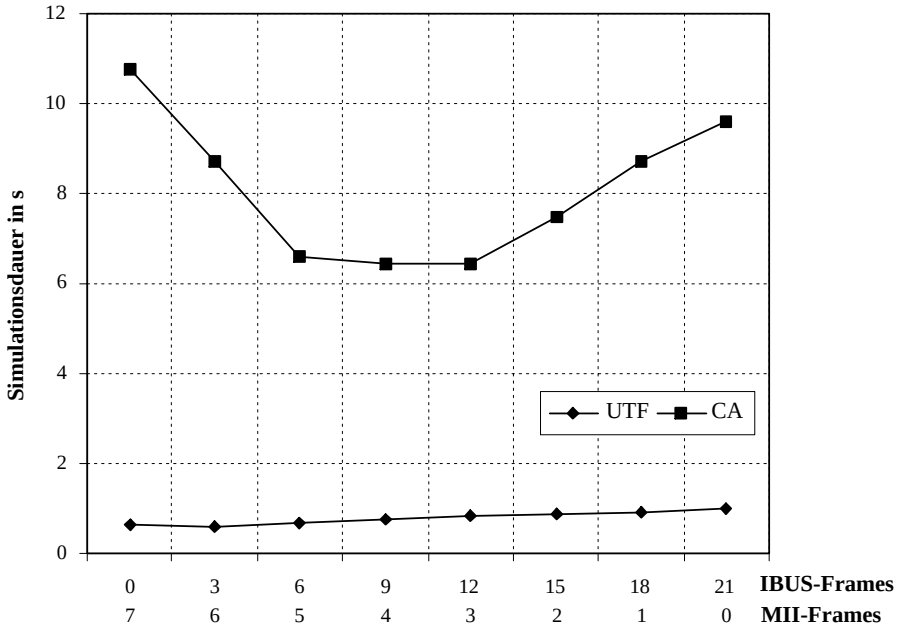

Abbildung 4.5: Perfomancevergleich zwischen CA- und UTF-Modell

in der Abbildung 4.5 dargestellt. Die Zahlenwerte auf der Abszisse sind so gewählt, dass die Anzahl der zu übertragenden Bytes eine Konstante ist, wobei die IBUS-Stimulidaten von links nach rechts und MII-Stimulidaten von rechts nach links zunehmen. Ein MII-Frame von 1600 Bytes entspricht etwa drei maximal große IBUS-Frames mit einer maximalen Größe von 512 Bytes. Daraus ergibt sich der Inkrement 3 bei den IBUS-Frames.

Eine Analyse des Diagramms bestätigt den erwarteten Geschwindigkeitsvorteil des abstrakten Modells, dessen Simulationsdauer durchwegs unter einer Sekunde liegt. Das Minimum des CA-Modells ergibt sich dadurch, dass bei gleicher Auslastung durch IBUS- und MII-Frames die Parallelität in der Verarbeitung der Daten am Größten ist. Die geringere Steigung auf der IBUS-dominierten Seite zeigt, dass das IBUS-Interface weniger aufwendig modelliert wurde als das MII: Die Daten werden im IBUS-Interface in einer Struktur, beim MII hingegen als einzelne Nibbles übergeben.

4.3 Verfeinerung der Schnittstellen

Nachdem im vorherigen Kapitel die Funktionalität der Blöcke beschrieben wurde, werden im nächsten Schritt die Schnittstellen zwischen diesen Blöcken, im speziellen der IBUS, die i486-Schnittstelle und das MII, verfeinert.

4.3.1 IBUS

Am IBUS werden die Protokoll- und Nutzdaten bitseriell mit einer Taktrate von 50 MHz übertragen. Ein Nachrichtenpaket besteht aus:

- einer Start-Rahmenerkennung (Start Flag),
- der Steuerinformation (Header),
- im Falle eines Datenpakets den Nutzdaten (Payload) und
- einer Ende-Rahmenerkennung (End Flag).

Die Übertragung der Daten erfolgt nach einem Handshake-Verfahren. Ein Datenpaket wird erst dann abgeschickt, wenn sichergestellt ist, dass sowohl die benötigten Datenwege frei sind als auch der Empfänger aufnahmebereit ist. Das Handshaking beginnt mit einer *Request*-Nachricht vom Absender an den Empfänger, der mit einer *Answer-Acknowledge*-Nachricht antwortet, sobald er für die Aufnahme des Datenpakets bereit ist. Die Answer-Acknowledge-Nachricht wird zum Sender zurückgeleitet und bewirkt dort das Absenden des Datenpakets. Falls der adressierte Empfänger ein Datenpaket auf absehbare Zeit nicht aufnehmen kann, beantwortet er eine Request-Nachricht mit einer *Answer-Refuse*-Nachricht. Der Absender reagiert auf den Empfang einer Answer-Refuse-Nachricht mit der Aussendung einer *Release-Request*-Nachricht.

Die Modellierung des IBUS soll nicht bitgenau sein – der IBUS ist ja auch keine Schnittstelle des IO-Controllers – , trotzdem ist eine Verfeinerung notwendig, denn im abstrakten Modell wurde auf das Protokoll noch keine Rücksicht genommen.

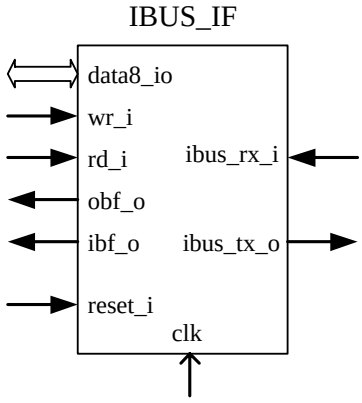

Abbildung 4.6: Das IBUS-Modul

Die Implementierung in SystemC wurde so gewählt, dass auf den beiden IBUS-Leitungen ibus_tx und ibus_rx nicht Bitströme, sondern Strukturen, die komplette IBUS-Rahmen beschreiben, transportiert werden. Das zeitliche Verhalten wird nachgebildet,

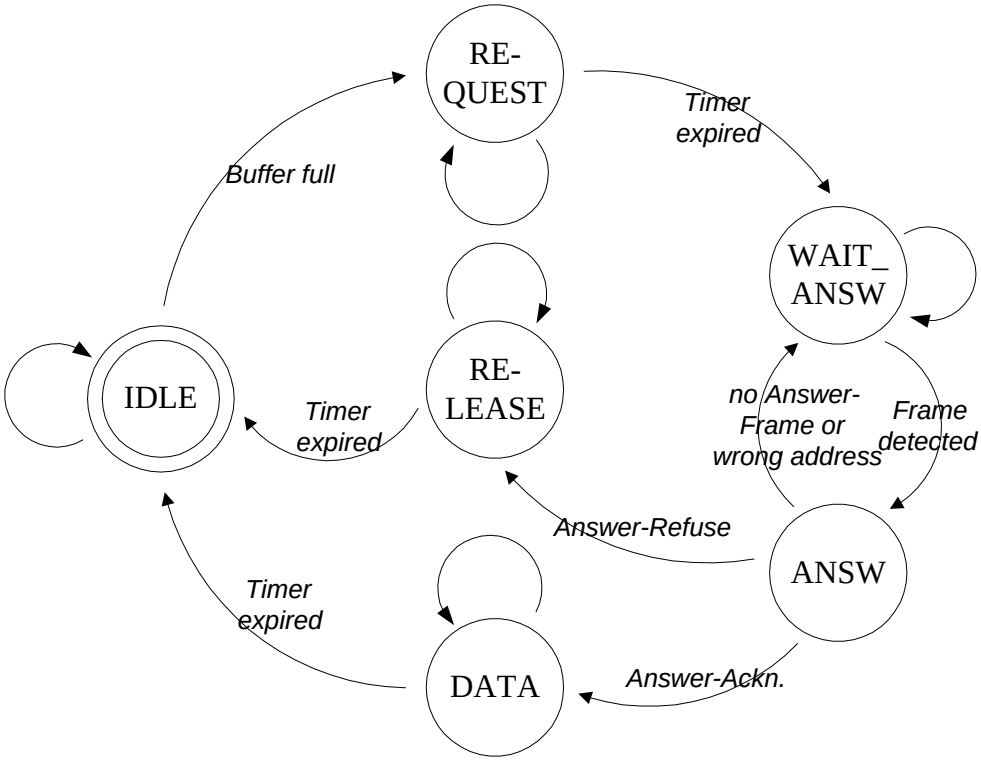

Abbildung 4.7: Zustandsdiagramm des IBUS-Sendeteils

indem diese Strukturen genau so lange an den Signalen liegen, wie der Bitstrom des entsprechenden Rahmens dauert. Das reduziert die Simulationsdauer erheblich.

Das IBUS-Interface wurde als Modul realisiert, dessen Ein- und Ausgabeports in der Abbildung 4.6 dargestellt sind. Es verwendet Handshaking, um die zu übertragende Datenstruktur in das Modul einzuspeisen.

Das IBUS-Modul besteht im Wesentlichen aus zwei Zustandsautomaten, je einen für Senden und Empfang. Die Zustandsdiagramme dafür sind in den Abbildungen 4.7 und 4.8 dargestellt.

Die Verifikation des Modells erfolgte durch Simulation einer Testbench mit zwei IBUS-Modulen, die durch Auskreuzen der ibus_rx- und ibux_tx-Leitungen miteinander verbunden wurden. Ein beispielhafter Kommunikationsablauf ist in der Abbildung 4.9 dargestellt. Dabei bedeuten die Zahlen bei den IBUS-Paketen: 1...Request, 2...Answer, 3...Daten, 4...Release-Request.

Wie man aus der Abbildung 4.9 erkennt, können Answer-Frames auch in den laufenden Datenstrom eingefügt werden – eine Notwendigkeit, um Deadlocks zu vermeiden. Das Modell hat einen Sendepuffer und zwei Empfangspuffer. Wenn also wie in der Abbildung ein drittes Datenpaket an einen Empfänger kommt, ohne vorheriges Auslesen eines Puffers, antwortet dieser mit einer Answer-Refuse-Nachricht. Der Initiator bricht

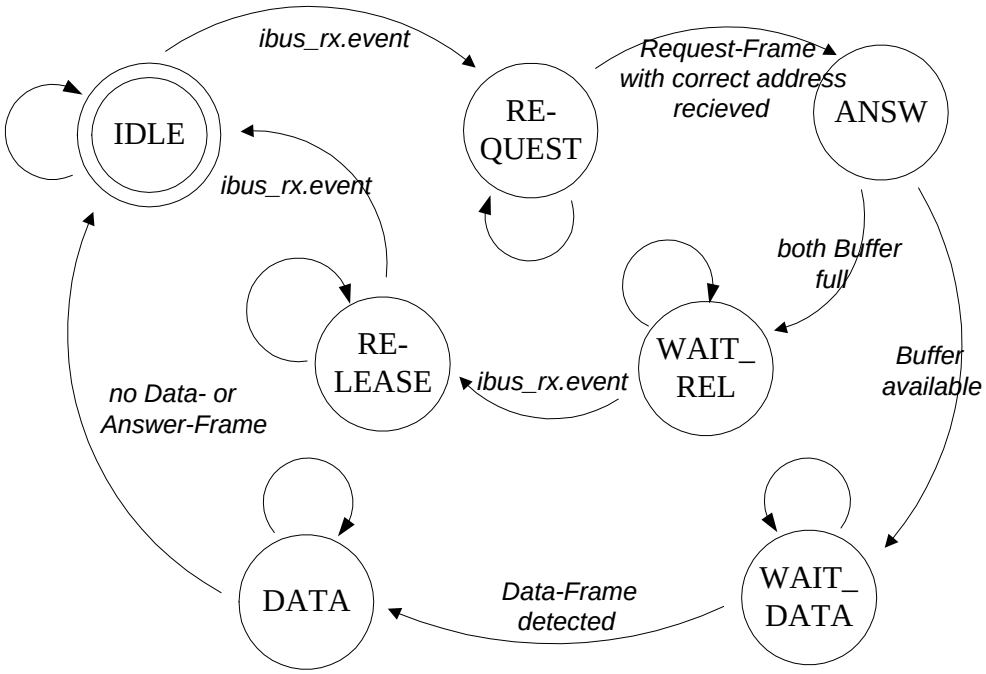

Abbildung 4.8: Zustandsdiagramm des IBUS-Empfängerteils

daraufhin mit einer Release-Request-Nachricht ab. Das wird solange wiederholt, bis ein Puffer ausgelesen wird und frei für neue Daten ist.

4.3.2 i486-Interface

Die Schnittstelle des ATMC zum IO-Controller ist eine Bus-Schnittstelle, bei der die Steuerleitungen und Adressen unidirektional sind und der Datenbus bidirektional ist. Sie ist im Wesentlichen einem i486-Prozessor-Interface ähnlich.

Die Datenbusbreite beträgt 32 Bit. Der Adressraum umfasst 4 GByte (30 Adress-Signale: A31...A2). Von den 86 Signalleitungen werden nicht alle modelliert, da Burst-Zugriffe der Einfachheit halber nicht ermöglicht werden. Pro Datenbyte existiert ein Signal für ein Paritätsbit und eines für die Aktivierung des Datenbytes (Byte Enable). Diese Signale werden ebenfalls nicht modelliert, da einerseits Störungen an der Schnittstelle nicht berücksichtigt werden und andererseits nur Daten mit voller 32-Bit Breite übertragen werden. Die Betriebsfrequenz beträgt 25 MHz.

Das Timing der Schnittstelle ist in der Abbildung 4.10 dargestellt. Zu Beginn des ersten Taktes eines jeden Zugriffes aktiviert der IO-Controller die gültigen Adressen, das Adress-Strobe-Signal ads_n und die Schreib/Leseleitung wr_n. Bei Schreibzugriffen wird wr_n auf 1 gesetzt und einen Takt später werden zusätzlich die Daten angelegt. Bei Lesezugriffen muss wr_n 0 sein und die Daten werden mit der Aktivierung von rdy_n

Abbildung 4.9: Simulationsergebnisse des IBUS

eingelesen. In beiden Fällen erfolgt der Abschluss des Zugriffes durch die Deaktivierung von rdy_n.

Der IO-Controller ist der Master der Schnittstelle. Die Initialisierung des IO-Controllers erfolgt jedoch durch den ATMC, indem nach dem Reset während der ersten beiden Aktivierungen des Signals *AR* (Attention Request) die Setup-Daten an den Datenbus gelegt werden. Beim ersten AR werden allgemeine Hardware-Konfigurationsdaten und beim zweiten AR die Adresse des ATM Control Blocks (kurz ACB), der den Aufbau und die Position der verketteten Listen im Speicher des ATMC beschreibt[2], übertragen.

Der im Listing 4.3 dargestellte Prozess der i486-Schnittstelle im ATMC wurde zu dem taktzyklusgetreuen Modell im Listing 4.4 verfeinert. Da die untersten zwei Adressbits nicht mit übertragen werden, muss die erhaltene Adresse um zwei Positionen nach links geschoben werden. Dabei ist zu beachten, dass diese Adressen identisch mit jenen im Speicher der Simulationsmaschine sind. Wenn also beim zweiten AR dem IO-Controller die Adresse von tx_frame_ptr mitgeteilt wird, ist das die tatsächliche Adresse in der Simulationsmaschine. Das ermöglicht die im Listing gezeigte kompakte Codierung.

[2]Der Speicherorganisation im ATMC ist wesentlich komplizierter als in Kapitel 4.1 dargestellt. In der Realisierung wird daher lediglich die Adresse von tx_frame_ptr übergeben.

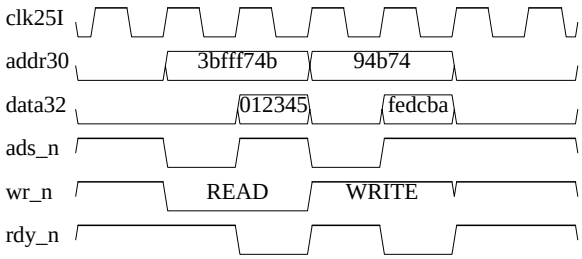

Abbildung 4.10: Simulationsergebnisse des i486-IF

Listing 4.4 Der verfeinerte i486_slave-Prozess

```
void mbdatm_m::i486_slave_proc(){
   // initialize ...
   // reset ...

   while (true) {
      wait_until(ads_n_i.delayed() == 0);
      addr = addr30_i.read();
      mp = mem_ptr(addr << 2);
      rdy_n_o = 0;
      if (wr_n_i){
         wait();
         data = data32_io.read();
         // write data into memory
         *mp = data;
      }
      else {
         // read data from memory
         data32_io = *mp;
         wait();
         data32_io = 0;
      }
      rdy_n_o = 1;
   }
}
```

4.3.3 Media Independent Interface (MII)

Das MII entspricht dem Standard IEEE802.3u mit jeweils 4 Bit parallelen Ein- und Ausgängen, synchron zu einem 25 MHz Eingangstakt, mit Enable und einem Signal für die Kollisionserkennung. Die Schnittstellensignale sind aus der Abbildung 4.12 ersichtlich.

Die Funktionalität dieser Schnittstelle ist sehr einfach. Der Empfang von Rahmen startet mit der Aktivierung des Signals mii_en_i. Solange dieses Enable-Signal aktiv ist, werden die am Bus mii_data4_i angelegten Daten synchron eingetaktet. Ausgangsseitig ist das gleiche Verhalten mit den Signalen mii_en_o und mii_data4_o vorgesehen. Der Einfachheit halber sollen nur volle Frames mit 400 Doppelwörtern berücksichtigt werden, d.h. die Enable-Signale des MII sind immer für 3200 Takte aktiv.

4.4 Verfeinerung IO-Controller

Nachdem die Schnittstellen genauer beschrieben worden sind, erfolgte die Verfeinerung des internen Aufbaus des IO-Controllers mit dem Resultat einer hardwarenahen, taktzyklengetreuen Beschreibung. Da die angrenzenden Schnittstellen i486 und MII mit jeweils einem eigenen Takt asynchron zueinander betrieben werden, ist eine Zwischenspeicherung notwendig, die durch FIFOs realisiert wurde, denen beide Takte zugeführt wurden. Die Umsetzung der 4 Bit Daten des MII auf die Bitbreite 32 der i486-Schnittstelle und umgekehrt erfolgt durch Multiplexer bzw. Demultiplexer. Schließlich sind noch Einheiten notwendig, die die i486-Schnittstelle betreiben. Die Abbildung 4.11 stellt den in SystemC codierten Aufbau in grafischer Form dar, wobei Rechtecke mit durchgezogenen Linien Submodule und jene mit strichlierten Linien Prozesse darstellen.

Der Multiplexer *mux* ist ein Modul, welches aus einem Prozess besteht, der bei aktivem Enable-Signal die 32 Bit Eingangsdaten übernimmt und in acht Nibbles aufteilt, die er hintereinander in acht Takten in Richtung MII ausgibt. Das Enable-Signal am Ausgang wird synchron zu den Daten aktiviert. Im Multiplexer findet also eine Serialisierung statt. Für die Parallelisierung ist das Modul *shifter* zuständig. Es fügt die vom MII kommenden Nibbles in acht Takten zu einem 32 Bit Wort zusammen, das synchron zur Aktivierung des Enable-Signales an den Ausgang geschrieben wird. Die beiden FIFO's *in_fifo* und *out_fifo* sind Zwischenspeicher, die jeweils mit den Takten beider Schnittstellen versorgt werden, um eine taktsynchrone Übernahme der Daten an den Ein- und Ausgangsseiten zu gewährleisten.

Der Prozess *control_write* steuert den Ablauf beim Schreiben des Controllers zum MII. Er liest periodisch die Speicherstelle TxFramePtr aus (Scanning). Ist deren Inhalt ungleich Null, d.h. die Software hat Daten hinterlegt, wird der Frame ausgelesen und

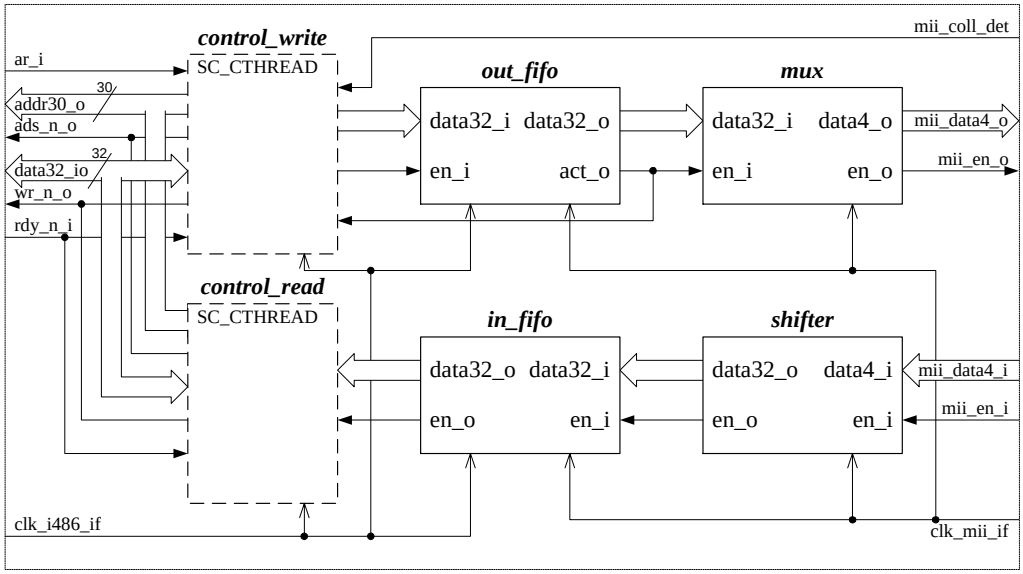

Abbildung 4.11: Aufbau des IO-Controllers

den nachfolgenden Bausteinen (FIFO, Multiplexer) übergeben, die die Ausgabe auf das MII durchführen. Solange der Zeiger auf den nächsten Frame ungleich Null ist, wird dieser Vorgang wiederholt. Der Prozess *control_read* steuert den Ablauf beim Empfang des Controllers vom MII. Nach der Eintaktung und Umsetzung (Shifter) laufen die Daten in ein FIFO. Das veranlasst den Controller dazu, aus dem RxPtrArray einen von der Software bereitgestellten Zeiger auf einen freien Puffer zu lesen und diesen mit den Daten aus dem FIFO zu füllen.

Eine Klasse namens Semaphore dient wie der Name schon sagt dazu, den Zugriff auf die i486-Schnittstelle zu koordinieren und so gleichzeitige Aktivitäten der Prozesse control_read und control_write an der Schnittstelle auszuschließen.

4.5 Simulation

Die in den Kapiteln 4.3 und 4.4 behandelten Teile sind die Bausteine des Gesamtsystems, das in der Abbildung 4.12 dargestellt ist. Da der IBUS eine wesentliche Komponente in diesem System ist, wurde durch Hinzufügen eines Tracers an den beiden IBUS-Leitungen, der IBUS-Pakete protokolliert, eine weitere Kontrollmöglichkeit neben den Ausgaben der Stimuligeneratoren geschaffen.

Die zur Ansteuerung des IBUS bzw. MII vom Stimuligenerator eingelesenen Dateien können einfache Befehle enthalten. Folgende Befehle werden vom IBUS-Generator akzeptiert:

- `send` *data...*

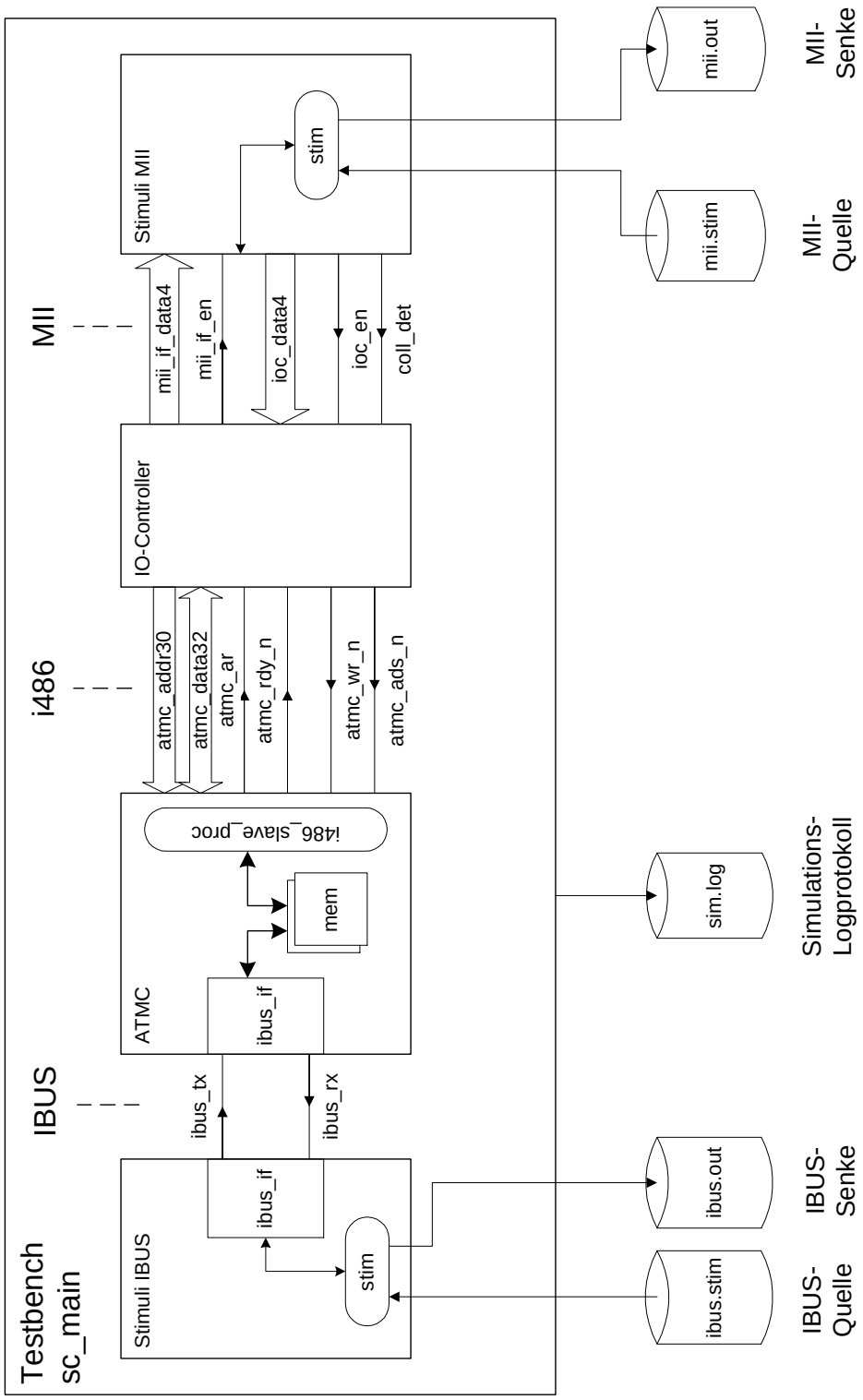

Abbildung 4.12: Verfeinertes Gesamtsystem

mit der Angabe von max. 512 Bytes für die zu sendenden Daten

- `sendpckt` *length*
  Paket mit angegebener Länge wird gesendet
- `set_request_type` *req_type*
  Einstellung des Request-Typs für folgende Übertragungen
- `set_dest_addr` *dest_addr*
  Einstellung der Zieladresse für folgende Übertragungen
- `wait` *time*
  wartet entsprechende Zeit in ns
- `wait_for_data`
  wartet auf den Empfang von Daten

Folgende Befehle werden vom MII-Generator akzeptiert:

- `send` *data...*
  mit der Angabe von max. 400 Doppelwörtern für die zu sendenden Daten - die Daten werden immer zu einem vollen Paket mit steigenden Dateninhalten aufgefüllt
- `collision`
  aktiviert das Kollisions-Signal für einen Taktzyklus
- `wait` *time*
  wartet entsprechende Zeit in ns
- `wait_for_data`
  wartet auf den Empfang von Daten

In der Loggdatei werden wie bei der Simulation des Systems auf abstrakter Ebene (Kapitel 4.2) bestimmte Zustände der Prozesse mit einem Zeitstempel protokolliert.

Ein Problem bei der Simulation mit SystemC ist, dass man im Normalfall keine Information über den Simulationsfortschritt hat, wenn nicht der Inhalt einer Loggdatei permanent überprüft wird oder Zeitstempel bzw. Zustände direkt auf die Konsole ausgegeben werden[3]. Deshalb wurde eine Klasse entwickelt, die Methoden zur Visualisierung des Simulationsfortschrittes und zur Berechnung der Simulationsperformance enthält. In der Abbildung 4.13 ist ein Terminal nach einer Simulation über 1 ms dargestellt. Während der Simulation wächst der durch '#'-Zeichen gebildete Balken von links

[3]Man erkennt so z.B. auch nicht, ob sich die Simulation in einer Endlosschleife (Threads oder CThreads ohne wait()-Kommando) festfährt.

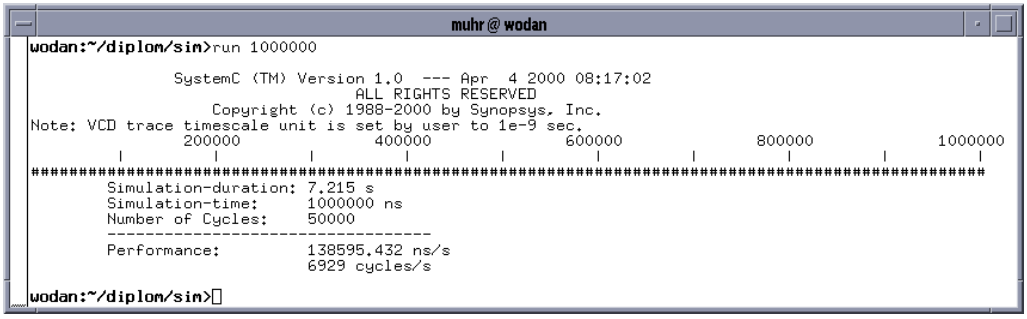

Abbildung 4.13: Ausgabe während der Simulation

nach rechts und zeigt somit die aktuelle Zeit an. Am Ende werden Informationen über die Simulationsdauer und die Simulationsgeschwindigkeit ausgegeben.

Um eine von der Komplexität des Systems unabhängige, universell einsetzbare Ausgabe zu erhalten, die die Simulation nicht belastet, kann man nicht einfach die Takte zählen oder die aktuelle Simulationszeit laufend abfragen (Polling). Es werden daher die betriebssystemeigenen Zeitgeber benutzt, um ein periodisches Signal (1 Hz) zu erzeugen, dessen Signalbehandlungsroutine die Simulationszeit ausliest und die Balkenanzeige aktualisiert.

Die Funktionalität des entwickelten Systems wurde durch mehrere Simulationen verifiziert, von denen hier eine kurz erläutert werden soll. Die Stimulis von beiden Seiten (IBUS und MII) sind im Listing 4.5 abgedruckt. Beim IBUS werden nach der Einstellung von Request-Typ und Zieladresse mehrere Pakete mit der maximalen Länge von 512 Bytes gesendet. Es sind zumindest 4 Pakete notwendig, um eine Übertragung zum MII zu ermöglichen, da nur volle, aus 400 Doppelwörtern bestehende Rahmen bearbeitet werden. Auf der Gegenseite werden zwei volle, durch 100μs getrennte, MII-Rahmen eingespeist. Die Waveforms sind in der Abbildung 4.14 dargestellt.

Listing 4.5 Inhalt der Stimulidateien

```
ibus.stim:                mii.stim:

set_request_type 1        send 80000001
set_dest_addr 2           wait 100000
sendpckt 512              send 80000001
sendpckt 512
sendpckt 512
sendpckt 512
sendpckt 512
sendpckt 512
```

Neben den Waveforms können auch die Ausgabedateien der Stimuligeneratoren, aus denen Ausschnitte im Listing 4.6 gezeigt sind, zur Verifikation herangezogen werden. Durch die Analyse von Waveforms erhält man einen guten Überblick darüber, was zu

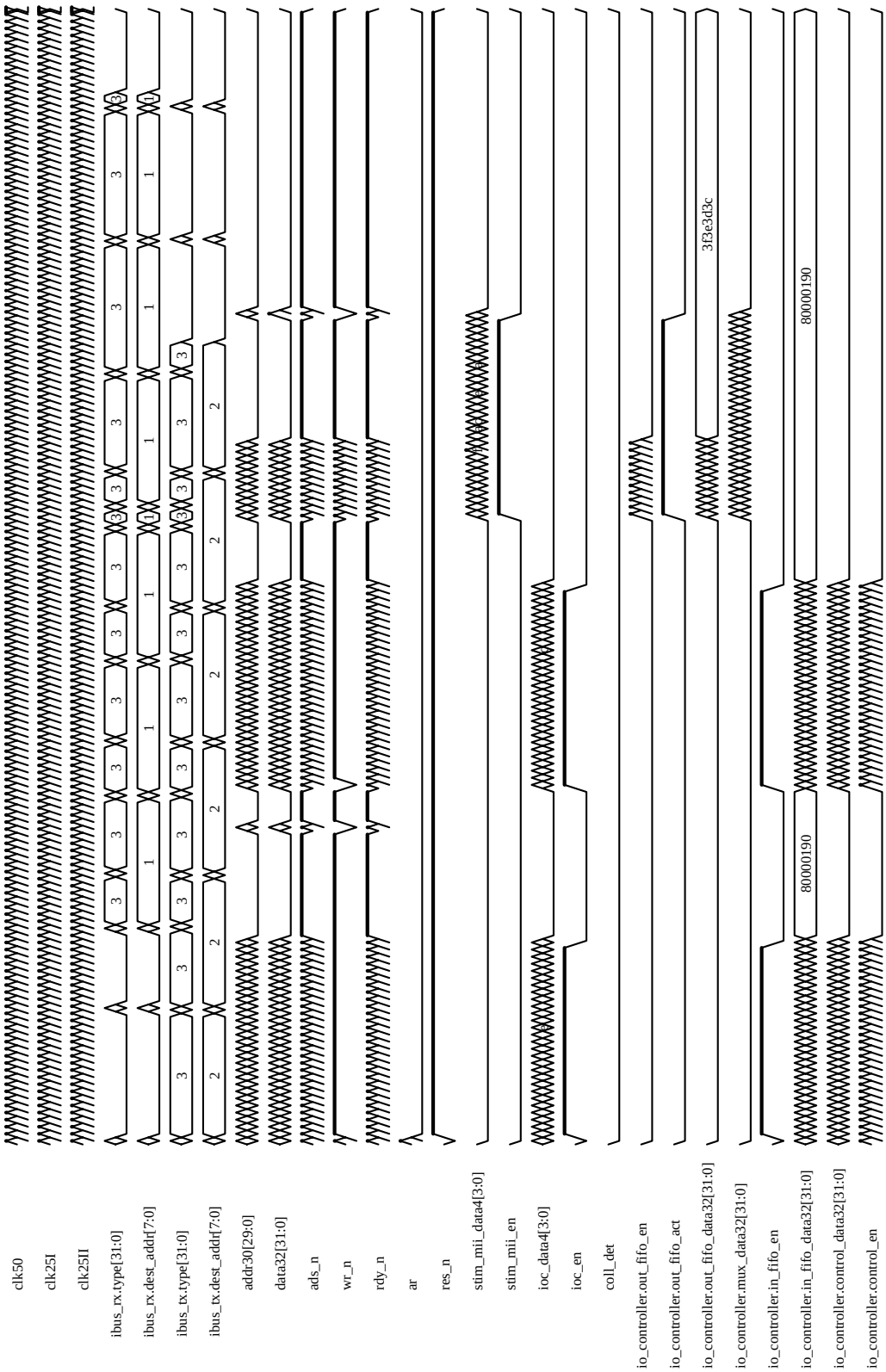

Abbildung 4.14: Waveforms der Simulation

welchem Zeitpunkt passiert. Eine genaue Überprüfung der Daten ist aber speziell bei derartigen datenintensiven Systemen zu aufwendig. Die Ausgabedateien erlauben hingegen eine automatische Verifikation durch Programme, die nach der Simulation einen Vergleich mit den Stimulidateien durchführen.

Listing 4.6 Ausschnitt aus den erzeugten Ausgabedateien

```
ibus.out:
0.000221580 data_pckt:  dest_addr = 1, req_type = 1
          01 00 00 80 02 00 00 80 03 00
          00 80 04 00 00 80 05 00 00 80
          06 00 00 80 07 00 00 80 08 00
          00 80 09 00 00 80 0a 00 00 80
          0b 00 00 80 0c 00 00 80 0d 00
          ...

mii.out:
0.000401030 data:
00102030 40506070 8090a0b0 c0d0e0f0
01112131 41516171 8191a1b1 c1d1e1f1
02122232 42526272 8292a2b2 c2d2e2f2
03132333 43536373 8393a3b3 c3d3e3f3
...
```

Bei der erfolgreichen Simulation sind alle Datenpakete, die an den Stimuli-Generatoren eingespeisten wurden, den kompletten Datenpfad durchlaufen und vollständig am anderen Ende der Übertragungskette angekommen.

4.6 Cosimulation mit Hardwarebeschreibungssprachen

Einer der wichtigsten Punkte bei der Überlegung, SystemC in zukünftigen Projekten miteinzubeziehen, ist die Wiederverwendbarkeit von bestehendem Code. Ein Übergang auf SystemC wird nicht schlagartig erfolgen. Es ist daher notwendig, eine gemeinsame Simulation von SystemC-Modulen mit VHDL- bzw. Verilog-Beschreibungen zu ermöglichen. Im Folgenden werden konkrete Ansätze präsentiert, wie die Cosimulation mit VHDL durchführbar ist.

4.6.1 Kommunikationskonzept

Da zum Zeitpunkt des Entstehens dieser Arbeit keine Werkzeuge zur Cosimulation zur Verfügung standen, wurden eigene Konzepte entwickelt. Das zu verifizierende System aus den vorangegangenen Kapiteln wurde an der i486-Schnittstelle aufgetrennt. Der IBUS und der ATMC aus der Abbildung 4.12 wurden in VHDL implementiert und der Rest, bestehend aus IO-Controller und MII, wurde aus der SystemC-Beschreibung übernommen. Die beiden Teile wurden separat simuliert.

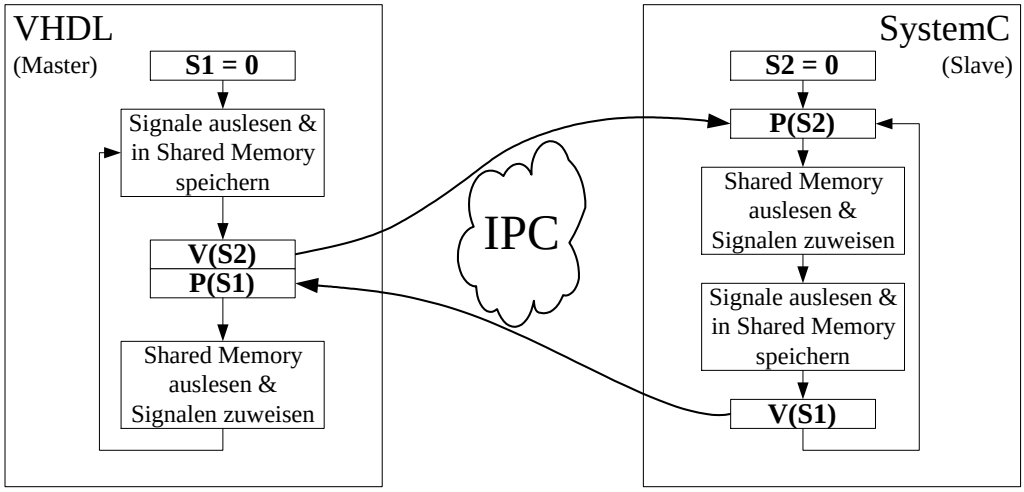

Abbildung 4.15: Kommunikationsablauf

Durch Interprozesskommunikation (engl. Abkürzung IPC) des Betriebssystems können zwei getrennte Simulationen synchronisiert werden. Der Austausch der Information über die Schnittstellensignale erfolgt mittels *Shared Memory*. Das ist ein vom Betriebssystem verwalteter Speicherbereich, der von mehreren Programmen bzw. Prozessen verwendet werden kann. Zur Synchronisation des Zugriffs auf diesen Speicher werden zwei *Semaphore* verwendet, die den Programmfluss mit den üblichen P- und V-Operationen[4] steuern [13]. Der Ablauf der Kommunikation zwischen den beiden Simulationen ist aus der Abbildung 4.15 ersichtlich.

Der VHDL-Simulator, der als Master fungiert, liest die Schnittstellensignale aus und überträgt deren Werte in das Shared Memory. Dann stößt er das SystemC-Programm als Slave mit der V-Operation an. Dieses aktualisiert die Signale, simuliert und übergibt die neuen Signalwerte wieder an den VHDL-Simulator zurück. Dieser Vorgang wird bei jeder positiven Flanke des Schnittstellentaktes wiederholt. Dazwischen laufen die Simulationen parallel ab.

Diese Variante der Cosimulation ist eine Vereinfachung für synchrone Systeme und kann so nicht generell angewendet werden, da nur ein Abgleich der Signale pro Takt erfolgt. Wäre mehr als ein Abgleich erforderlich, damit die Schnittstellensignale stabil sind, müssten die P- und V-Operationen iterativ angewendet werden bis die Stabilität erreicht ist. Die Erweiterungen diesbezüglich wären gering und ohne Probleme durchführbar. Bei taktzyklengetreuen Beschreibungen, bei denen Änderungen zufolge eines Signalwechsels meist erst im nächsten Takt modelliert werden, stellt dies aber

[4]Die P-Operation wartet, bis der Wert der zugehörigen Semaphorvariablen größer als Null ist und verringert dann deren Wert um Eins. Die V-Operation erhöht den Wert der zugehörigen Semaphorvariablen um Eins und ermöglicht somit die Terminierung von P-Operationen, die dem selben Semaphor zugeordnet sind.

ohnehin keine allzu große Einschränkung dar.

4.6.2 VHDL-Teil

HDL-Simulatoren werden in der Regel mit einer C-Schnittstelle ausgestattet, die das Einbinden von Modellen, die in C geschrieben sind, erlaubt. Die in VHDL standardisierte Schnittstelle heißt *Foreign Language Interface* (kurz FLI). Durch das FLI können Architekturen und Unterprogramme von VHDL durch C-Modelle ersetzt werden. Als VHDL-Simulator wurde Modelsim von Mentor Graphics verwendet.

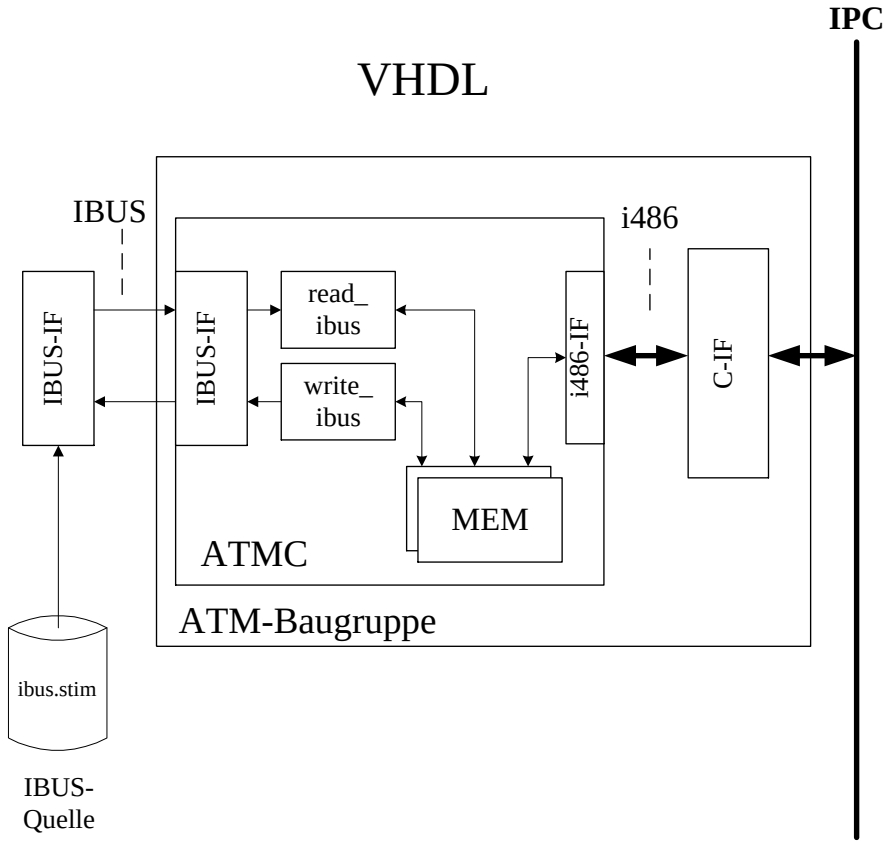

Abbildung 4.16: VHDL-Seite zur Cosimulation

In der Abbildung 4.16 ist der VHDL-Teil der Cosimulation dargestellt. Da das bestehende VHDL-Modell des ATMC nicht verfügbar war und die Einarbeitung für diesen komplexen Baustein ohnehin zu umfangreich gewesen wäre, wurde ein stark vereinfachtes Modell implementiert, das mit jenem in SystemC beschriebenen, verfeinerten Gesamtsystem vergleichbar ist. Ein vorhandenes IBUS-Modell wurde einerseits als Stimuligenerator und andererseits in modifizierter Form im ATMC eingebaut. Der Speicher im ATMC wurde als globales Feld definiert, was in VHDL93 standardisiert ist. Dadurch

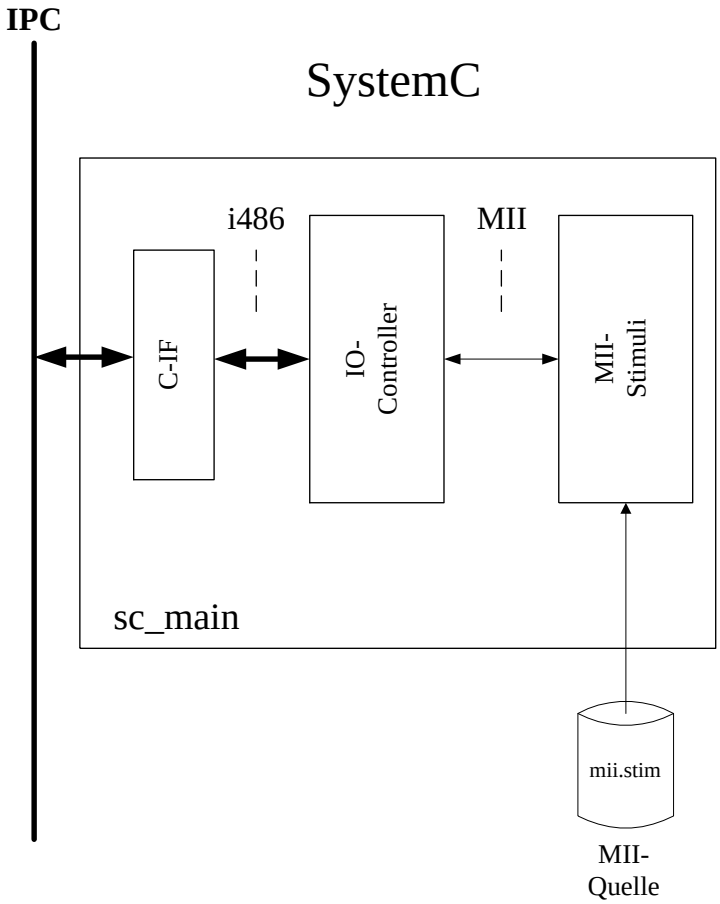

Abbildung 4.17: SystemC-Seite zur Cosimulation

können die Prozesse read_ibus, write_ibus und i486-IF ohne viel Aufwand auf einen gemeinsamen Speicher zugreifen. Die Funktionalität dieser drei Prozesse ist der in Kapitel 4.2 beschriebenen äquivalent. Der Block C-IF ist in der oben beschriebenen Weise für die Interprozesskommunikation mit der SystemC-Simulation zuständig.

4.6.3 SystemC-Teil

Im SystemC-Programm wurde lediglich die Testbench `sc_main` verändert. Die Komponenten IBUS-Stimuligenerator und ATMC wurden entfernt und wie aus der Abbildung 4.17 ersichtlich durch Funktionen zur Interprozesskommunikation ersetzt.

4.6.4 Werkzeuge

Einige Firmen sind im Begriff, ihre bestehenden Werkzeuge um eine SystemC-Kompatibilität zu erweitern bzw. gänzlich neue Produkte entwickeln, die SystemC

unterstützen. Beim Entstehen dieser Arbeit[5] konnten drei Werkzeuge recherchiert werden, die eine Cosimulation von SystemC mit HDL unterstützen:

CoCentric System-Level-Design

Die CoCentric Produktfamilie von Synopsys enthält neben dem SystemC-Compiler auch eine Schnittstelle zur Cosimulation mit HDL-Simulatoren. Es werden der Verilog-Simulator VCS, Scirocco und der VHDL-Simulator von Modeltech unterstützt. Die Cosimulationsumgebung ist dem in den Abbildungen 4.16 und 4.17 gezeigten Aufbau sehr ähnlich. Wie im vorigen Kapitel wird die Interprozesskommunikation des Betriebssystems verwendet, um die VHDL- und die SystemC-Simulationen zu synchronisieren, wobei die SystemC-Simulation den Master der Cosimulation darstellt, d.h. die Testbench muss in SystemC realisiert sein. Der umgekehrte Fall der Einbindung von SystemC-Modulen in eine VHDL-Simulation, wie er im Rahmen dieser Arbeit behandelt wurde, ist in Planung.

Ein Nachteil ist, dass das Aufsetzen der Umgebung für solch eine Cosimulation einige nicht triviale Schritte erfordert. Unter anderem muss eine Anpassung der Datentypen erfolgen.

Visual SLD

SLD steht für System Level Design und ist ein Produkt von Innoveda[6]. Es handelt sich um eine grafische Umgebung, die die Entwicklung von Systemebene bis hinunter auf Gatterebene unterstützt und eine Cosimulation von HDL-, C- und SystemC-Blöcken erlaubt.

SystemModeler

Laut Kevin Kranen, Direktor der strategischen Planung bei Synopsys, war SystemModeler von Transmodeling[7] das erste SystemC-basierte Werkzeug auf dem Markt. Wie auch Visual SLD ist SystemModeler eine grafische Umgebung für den Entwurf und die Verifikation von Systemen auf allen Abstraktionsebenen. Besonders hervorzuheben ist das Konzept der verteilten, parallelen Simulation. Einzelne Blöcke – beschrieben mit SystemC oder HDL – können auf unterschiedlichen (ausgewählten) Workstations simuliert werden, wobei die Kommunikation zwischen den Workstations völlig transparent abläuft. SystemModeler unterstützt Cadence Verilog XL, Synopsys VCS und die Modeltech Simulatoren.

[5] Oktober 2000
[6] http://www.innoveda.com
[7] http://www.transmodeling.com

4.7 Synthese

4.7.1 Grundlagen

Die Synthese bei der Entwicklung von integrierten Schaltkreisen bedeutet im allgemeinen Sinn die Umsetzung einer Schaltungsbeschreibung von einer höheren Ebene auf eine implementierungsnähere Ebene. Am unteren Ende des Entwurfsablaufs ist die Logiksynthese angesiedelt, die Beschreibungen der Registerebene (*Register Transfer Level*, kurz: RTL) zu Gatternetzlisten detailliert, also Register, Addierer, Multiplizierer, FSM, Multiplexer usw. durch Logikgatter ersetzt. Dieser Schritt kann heute ohne Probleme automatisiert werden.

Weit kritischer hingegen ist der Übergang von Architekturen, deren Funktion durch Algorithmen beschrieben wird, auf eine Darstellung in der Registerebene. Diese sogenannte Architektursynthese[8] extrahiert den Zustandsautomaten, die Ein- und Ausgabeprotokolle und den Datenpfad aus einer Verhaltensbeschreibung, die sich insbesondere durch ihre Technologieunabhängigkeit auszeichnet. Mittels Werkzeugen, die eine Architektursynthese automatisiert durchführen, kann die Time-to-Market drastisch gesenkt werden [14]. Der Trade-Off zwischen Performance und Chipfläche kann durch globale Vorgaben vom Systemarchitekten optimiert werden.

4.7.2 SystemC-Compiler

Das derzeit einzige auf dem Markt erhältliche Werkzeug, das überhaupt eine Synthese von SystemC-Code erlaubt, ist der SystemC-Compiler aus der Produktfamilie CoCentric von Synopsys. Wie in der Abbildung 4.18 dargestellt, verarbeitet der Compiler SystemC-Quellcode mit Parametern für die Synthese und Bibliotheken zu Beschreibungen auf RTL-Ebene in verschiedenen Ausgabeformaten, die im weiteren zur Logiksynthese verwendet werden können. Die interaktive grafische Analyseumgebung namens BCView verarbeitet die zusätzlich ausgegebenen Report-Informationen und unterstützt den Entwickler bei der Verifikation der Systemvorgaben.

[8] Von Synopsys wird der Begriff Verhaltenssynthese gebraucht, der eigentlich nicht ganz richtig ist, weil für die Synthese eine taktzyklengetreue Beschreibung erforderlich ist, und nicht nur eine Verhaltensbeschreibung.

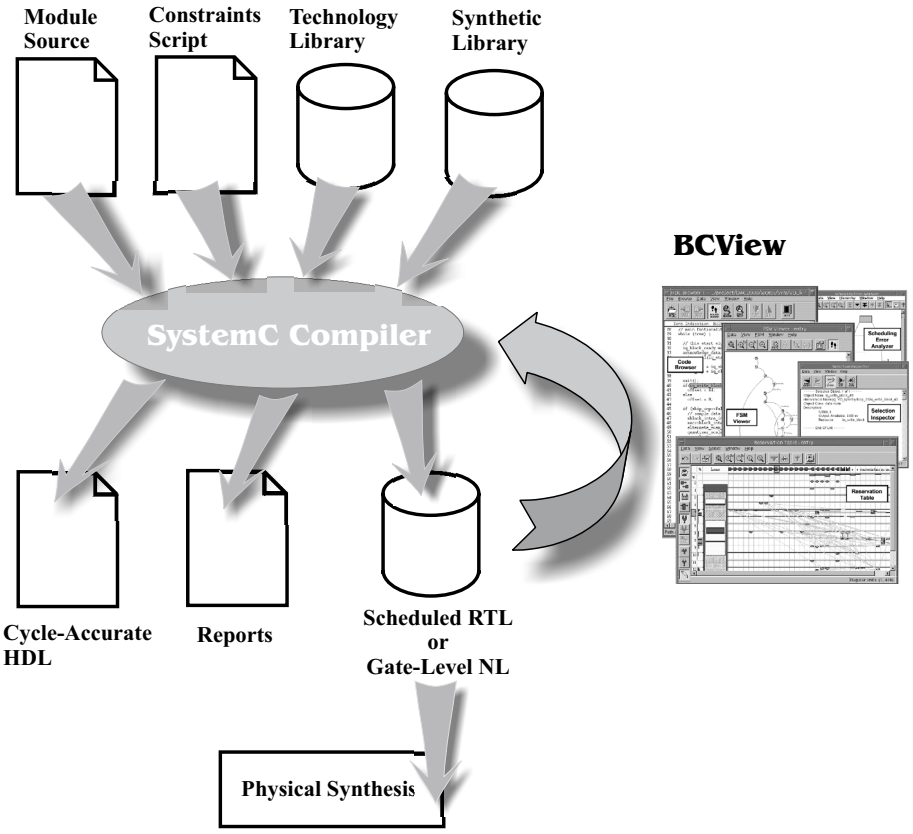

Abbildung 4.18: Ein- und Ausgabe des SystemC-Compilers

Die Abbildung 4.19 zeigt den Entwicklungsablauf bei der Verwendung des SystemC-Compilers. Ausgehend von einem funktionellen Entwurf, der in reinem C++ verfasst sein kann, erfolgt eine schrittweise Verfeinerung der Beschreibung durch den Entwickler, indem er SystemC-Konstrukte zur Modellierung von Kommunikation, Zeitverhalten, Speicher, usw. hinzufügt. Das Ergebnis ist eine taktzyklengetreue, implementierbare Beschreibung des Systems, die durch die automatische Verhaltenssynthese des SystemC-Compilers in synthetisierbare Netzlisten übersetzt wird.

Es werden drei verschiedene Formate unterstützt. Eine RTL-Netzliste, deren Format wahlweise VHDL oder Verilog sein kann, erlaubt die Logik-Synthese durch andere Werkzeuge für einen schnellen FPGA-Entwurf. Diese Netzliste ist zwar auch simulierbar, zur Steigerung der Simulationsperformance bei umfangreichen Systemen wird aber eine zusätzliche, flache Netzliste ohne Hierarchien erzeugt. Eine parametrisierte Netzliste im '.db'-Format dient der anschließenden Logik-Synthese durch den Synopsys Design Compiler, der letztendlich eine Gatternetzliste zur Integration auf einem Chip erzeugt.

Im Folgenden werden die wichtigsten Merkmale des SystemC-Compilers beschrieben. Bei der Übersetzung von Algorithmen in Hardware werden Variablen und Signale

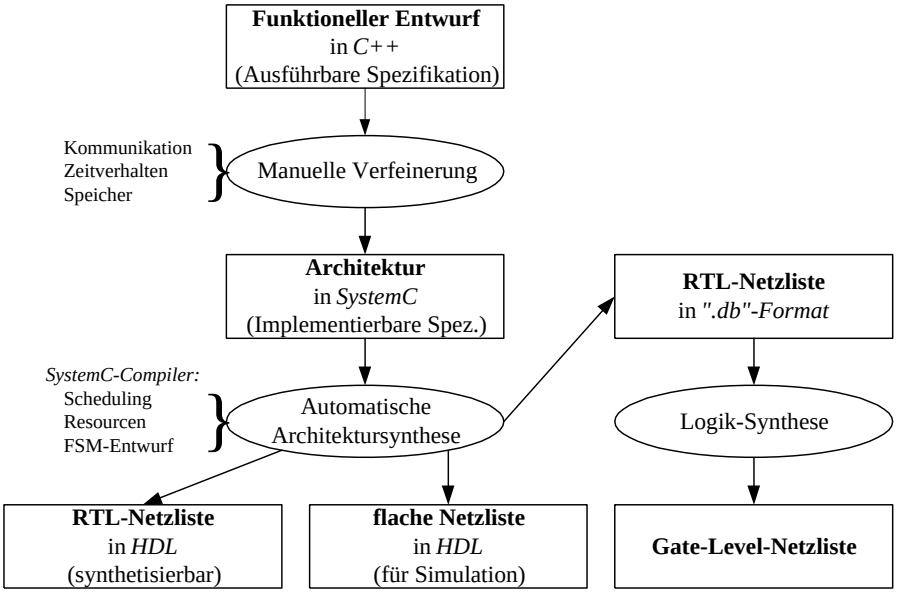

Abbildung 4.19: Entwicklungsablauf mit dem SystemC-Compiler

auf Register bzw. Speicher und Rechenoperationen auf entsprechende Hardwarekomponenten (Addierer, Multiplizierer, usw.) abgebildet. Durch Constraints kann gesteuert werden, ob und wie sich verschiedene Algorithmen gemeinsame Hardwarekomponenten teilen. Somit hat der Entwickler die Möglichkeit, die Qualität der Ergebnisse hinsichtlich Chipfläche und/oder Performance zu optimieren.

Auf die Zuweisung von Rechenoperationen, Ein- und Ausgabeoperationen sowie Speicherzugriffe zu den Taktzyklen kann ebenfalls durch Constraints wie Taktperiode, Latenzzeit, Durchsatz und Fläche eingegriffen werden. Operationen, die mehr als eine Taktperiode zur Beendigung benötigen, werden automatisch über mehrere Takte verteilt. Durch die Unterstützung von Pipelining, das die Eingabe von neuen Operanden vor dem Fertigwerden der laufenden Operation erlaubt, kann der Durchsatz des Systems drastisch erhöht werden.

Variablen sind üblicherweise nicht permanent aktiv. Eine Analyse der„Lebenszeit“ der Variablen wird durchgeführt, um jene Variablen, die nie gleichzeitig in Verwendung sind, in gemeinsamen Registern abzuspeichern. Dadurch wird die Anzahl der Register und in Folge die Chipfläche reduziert.

Nach der Auswahl der Hardwarekomponenten für den Datenpfad erfolgt die automatische Erstellung des Zustandsautomaten (FSM) durch die Verarbeitung der gewonnenen Informationen über den Kontroll- und Datenfluss. Das erspart die Zeit zur Neuimplementierung der Kontrolllogik bei Änderungen im Entwurf.

BCView

Die grafische Oberfläche BCView soll wie in der Abbildung 4.18 angedeutet den interaktiven und iterativen Prozess zwischen Compiler und Entwickler bei der Verhaltenssynthese fördern, indem die bei der Synthese erhaltenen Ergebnisse in übersichtlicher Form dargestellt werden. BCView besteht aus vier Fenstern:

- Quellcode-Betrachter
- Reservierungs-Tabelle
- FSM-Betrachter
- Anzeige für zusätzliche Informationen

Der Quellcode-Betrachter zeigt den SystemC-Code und ist mit allen anderen Fenstern verkettet. Wenn Objekte in den anderen Fenstern selektiert werden, werden auch die entsprechenden Anweisungen im Quellcode hervorgehoben.

Die Reservierungs-Tabelle stellt eine Fülle von Informationen dar. Die verwendeten Hardware-Komponenten werden über den Kontrollschritten in einem Raster angeordnet. Es ist somit gut ersichtlich, in welchen Zuständen welche Ressourcen gebraucht werden. Durch Pfeile werden die Datenabhängigkeiten zwischen den Operationen, den Registern und der Ein- und Ausgabe sichtbar gemacht. Detaillierte Informationen über Schleifen, Speicherzugriffe, Pipelining, Rechenoperationen, Register, Multiplexer u.a. werden in grafischer Form abgebildet.

Der FSM-Betrachter illustriert die generierte FSM in einem Zustandsdiagramm und erlaubt dem Entwickler, den Entwurf schrittweise durchzugehen und zu verfolgen, welche Ressourcen in einem bestimmten Zustand aktiv sind.

Die Anzeige für zusätzliche Informationen stellt Eigenschaften wie die eingenommene Fläche, die Geschwindigkeit, den Namen, die Technologe usw. der in den anderen Fenstern ausgewählten Objekte dar.

Codierstil

Eine vergleichsweise kleine Untermenge der Sprachdefinition von C++ und SystemC ist für die Synthese zugelassen. Im Folgenden sind die nicht synthetisierbaren Konstrukte aufgelistet:

- Anweisungen ohne Bedeutung für die Synthese wie `printf`, `cout`, Tracing von Signalen, Anweisungen zur Simulationskontrolle wie `sc_start()`, `sc_initialize()`
- Klassendefinitionen, abgeleitete Klassen, usw.

- Asynchrone Prozesstypen `SC_THREAD` und `SC_METHOD`[9]
- Globale Variablen
- Zeiger und der Dereferenzier-Operator (&) mit Ausnahme bei der Rückgabe von Funktionswerten
- Mehrere `watching`-Anweisungen, d.h. nur eine Reset-Leitung kann berücksichtigt werden
- Lokales Watching
- Rekursive Funktionsaufrufe
- Dynamische Speicherverwaltung, also Anweisungen wie `malloc`, `free`, `new`, `delete`
- Bidirektionale Ports durch `sc_inout<>`,
- Exception-Handling von C++ (`throw`, `catch`, `try`)
- C++-Gleitkommadatentypen `double`, `float`,

Als Datentypen sind vorzugsweise die SystemC-Datentypen zu verwenden, da die ganzzahligen Datentypen von C++ maschinenabhängig sind und bei geringen benötigten Bitbreiten überflüssige Hardware erzeugen.

Es können nur einzelne Module ohne Submodule jedoch mit mehreren Clocked Thread-Prozessen kompiliert werden. Die Funktionen solcher Prozesse beginnen mit den Reset-Anweisungen, welche mit dem `wait`-Befehl von der Endlosschleife abgegrenzt ist, in der die eigentliche Funktion realisiert ist. Die generellen Regeln sind:

- In jeder Schleife muss zumindest ein `wait` enthalten sein.
- Schreibzugriffe auf das selbe Port müssen zumindest durch ein `wait` getrennt sein.
- Es können nur synchrone Reset-Signale modelliert werden. Asynchrone Reset-Signale müssen explizit durch Compiler-Kommandos angegeben werden.
- Wenn eine Verzweigung (`if`- oder `switch`-Anweisung) eine `wait`-Anweisung enthält, müssen auch alle anderen Zweige zumindest ein `wait` enthalten.
- Der Ausstieg aus jeder Schleife von verschachtelten Schleifen kostet einen Taktzyklus.

[9] Method-Prozesse sind für RTL-Beschreibungen gedacht, aber in der aktuellen Compiler Version 2000.05-SCC1.0 noch nicht unterstützt.

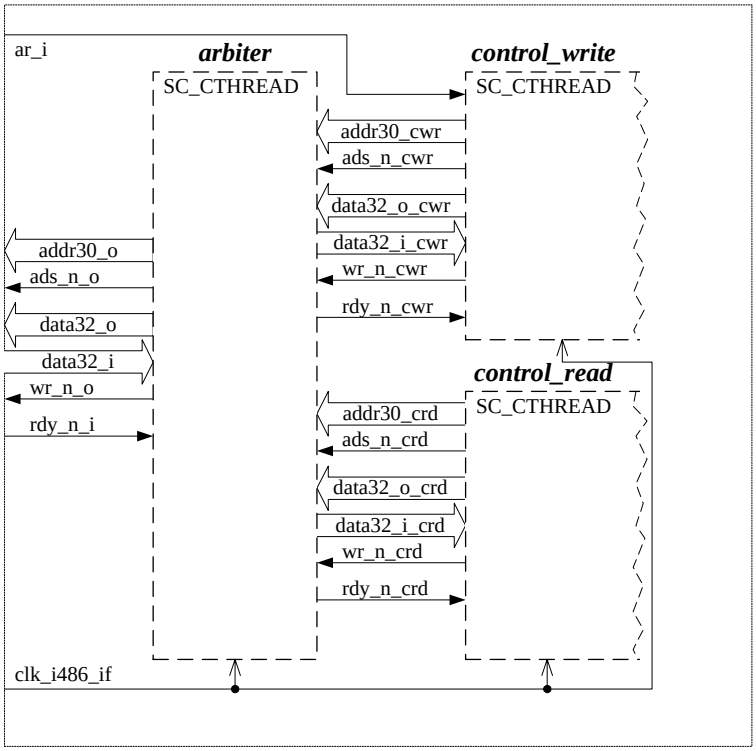

Abbildung 4.20: Ausschnitt aus IO-Controller: Arbiter-Prozess

Der Compiler erlaubt „*Loop Unrolling*“ und die Definition von „*Preserved Functions*“. Dadurch kann eine Funktion als Block definiert werden, der eine einzelne Operation darstellt und von anderen Werkzeugen (Module Compiler, Design Compiler) generierte Module aufnehmen kann.

Ein grafischer „*Memory Wrapper*“-Generator erleichtert das Einbinden von Speicherbausteinen.

4.7.3 Anwendung

Aufgrund der oben beschriebenen Beschränkungen bei der Synthese mussten, um den IO-Controller synthetisieren zu können, noch einige Veränderungen im Quellcode vorgenommen werden. Der 32 Bit Datenein- und Datenausgang der i486-Schnittstelle musste in ein Eingangsport und ein Ausgangsport aufgeteilt werden. Einige übriggebliebene Zeiger auf Speicherstellen mussten in den Hardware-Datentyp `sc_uint<32>` umgewandelt werden. Das Eingangssignal `coll_det`, das zur Kollisionserkennung dient, wurde deaktiviert, weil nur eine Watching-Anweisung erlaubt ist. Sämtliche Ausgaben auf den Bildschirm wurden mit Compiler-Direktiven versehen, um sie bei der Kompilierung zu unterdrücken.

Des weiteren wurden alle vier Submodule (in_fifo, out_fifo, mux, shifter) entfernt

und deren Funktionalität durch Prozesse im Modul des IO-Controllers realisiert. Da bei den beiden FIFO's mehrere Prozesse auf einen gemeinsamen Speicher zugegriffen haben, mussten hier „Shared Memories" eingesetzt werden.

Ein gröberes Problem betraf die i486-Schnittstelle. Da die beiden Prozesse *control_write* und *control_read* auf gemeinsame Ausgänge zugriffen, wurde wie in der Abbildung 4.20 dargestellt ein *Arbiter*-Prozess eingefügt, der den Zugriff auf die Schnittstelle kontrolliert. Er wurde so implementiert, dass Starvation ausgeschlossen ist, d.h. die beiden zugreifenden Prozesse haben auf jeden Fall die gleichen Rechte.

Als Ergebnis der Synthese zeigt das Listing 4.7 einen Ausschnitt aus dem Scheduling-Report.

Listing 4.7 Ausschnitt vom Scheduling-Report der Synthese

```
                  Behavioral Compiler (TM)
...
     Version 2000.11-PROD for sparcOS5 -- Oct 27, 2000
         Copyright (c) 1988-2000 by Synopsys, Inc.
                    ALL RIGHTS RESERVED
...
*********************************************
*  Summary report for process control_read:  *
*********************************************
------------------------------------------------------------
     Timing Summary
------------------------------------------------------------
 Clock period 20.00
 Loop timing information:
    control_read.............12 cycles (cycles 0 - 12)
        loop_304.............12 cycles (cycles 0 - 12)
            loop_305.........1 cycle  (cycles 0 - 1)
...
------------------------------------------------------------
     Area Summary
------------------------------------------------------------
 Estimated combinational area    3392
 Estimated sequential area       11931
 TOTAL                           15323

 14 control states
 20 basic transitions
 4 control inputs
 33 control outputs

------------------------------------------------------------
     Resource types
------------------------------------------------------------
Register Types
=======================================
1-bit register.....................2
9-bit register.....................1
32-bit register....................6

Operator Types
=======================================
(9->32)-bit RR_S_2P_1024_8_wrap.....1
(9_9->1)-bit DW01_cmp2..............1
(32_32->32)-bit DW01_add............1

I/O Ports
=======================================
1-bit input port...................2
1-bit registered output port.......2
30-bit registered output port......1
32-bit input port..................1
32-bit registered output port......1
...
```

Kapitel 5

Bewertung der Ergebnisse

Die folgenden Punkte bilden eine Zusammenfassung der Ergebnisse der in der Aufgabenstellung gestellten Fragen.

5.1 Entwurf eines Systems auf funktionaler und Hardware-Ebene

Im Kapitel 4 wurde demonstriert, wie SystemC genutzt werden kann, um Systeme auf verschiedenen Abstraktionsebenen zu modellieren. Ein Ausschnitt aus einem digitalen Sprachvermittlungssystem wurde nach den entsprechenden Vorgaben zuerst auf UTF-Ebene entworfen, um nach schrittweisen Verfeinerungen zu einem taktzyklengetreuen Modell auf Hardware-Ebene zu gelangen.

Die Beschreibung von parallelen sowie seriellen Schnittstellen (i486-Schnittstelle, IBUS) wurde durchgeführt. Der Entwurf von Zustandsautomaten wurde anhand des IBUS-Modells, einem Ausschnitt aus dem Gesamtsystem, gezeigt. Die Erstellung von Testbenches wird durch die Ein- und Ausgabefähigkeiten von C bzw. C++ gegenüber Hardwarebeschreibungssprachen erheblich erleichtert. Das Einlesen von Befehlsfolgen durch Stimuligeneratoren wurde in der Anwendung gezeigt. Dadurch können viele unterschiedliche Testfälle erzeugt werden, ohne den Quellcode des Entwurfs zu ändern und eine Neukompilierung zu erzwingen.

Ein Performancevergleich zwischen der UTF- und der CA-Beschreibung brachte die erwarteten Geschwindigkeitsvorteile bei der Simulation von Entwürfen auf UTF-Ebene. Der Einsatz von abstrakten Systembeschreibungen zu Beginn des Entwurfsablaufs bietet den zusätzlichen Vorteil, funktionale Fehler und Schwächen des Entwurfs in einem früheren Stadium zu entdecken, wo eine Verbesserung leichter und schneller durchzuführen ist, und somit die Time-to-Market, die Produktqualität und die Entwicklungskosten positiv beeinflusst.

5.2 Abstraktionsgewinn und Simulationsperformance

Aussagen von OSCI-Mitgliedern, wonach die SystemC-Simulation 50- bis 100-fach schneller als herkömmliche Verilog- und VHDL- Simulationen sind, konnten nicht verifiziert werden. Hier konnte lediglich ein Faktor 3 erreicht werden, wobei bei einer genauen Betrachtung auffällt, dass bei dem behandelten Beispiel die Abstraktionsebene der SystemC-Beschreibung zwar auch hardwarenahe ist, aber dennoch nicht in dem Maße wie die entsprechende VHDL-Beschreibung. Daraus ergibt sich die dreifache Geschwindigkeitszunahme bei der Simulation.

Es wäre auch kaum vorstellbar, dass C++-Compiler wie der *gcc* einen in dem Maß effektiveren Maschinencode bei annähernd gleicher Komplexität der Beschreibung erzeugen, als HDL-Compiler (vcom), die auf solche Simulationen optimiert sind. Im Gegenteil, eine exakte Umsetzung der VHDL-Anweisungen in entsprechende SystemC-Konstrukte hat wie in Kapitel 3.3 gezeigt sogar eine langsamere SystemC-Simulation zur Folge.

Der wesentliche Punkt bei solchen Vergleichen ist der Abstraktionslevel des Designs. Abstraktere Beschreibungen bedingen natürlich auch eine schnellere Simulation. Der Vorteil von SystemC liegt weniger in der verkürzten Simulationsdauer, sondern es unterstützt die Beschreibung auf abstrakterer Ebene, was sich schlussendlich positiv auf die Simulationsperformance auswirkt. Es hat macht daher wenig Sinn, ein D Flip Flop oder ein NAND-Gatter mit SystemC zu beschreiben.

5.3 Coverifikation mit Hardwarebeschreibungssprachen

Einer der wichtigsten Punkte bei der Überlegung, SystemC in zukünftigen Projekten mit einzubeziehen, ist die Wiederverwendbarkeit von bestehendem Code. Es ist daher notwendig, eine Umgebung für eine gemeinsame Simulation von VHDL bzw. Verilog mit SystemC zur Verfügung zu haben.

Da entsprechende Cosimulations-Werkzeuge beim Entstehen dieser Arbeit noch nicht verfügbar waren, wurde ein eigenes Konzept entwickelt, das die prinzipielle Machbarkeit der Cosimulation von SystemC mit HDL bestätigte. Dazu wurde ein Teil des Anwendungsbeispieles in VHDL codiert und mit dem Rest einer Coverifikation unterzogen.

Es stellte sich heraus, dass die CoCentric-Produktfamilie von Synopsys sehr ähnliche Ansätze bei der Cosimulation anwendet. Das Aufsetzen der Simulationsumgebung ist bei diesem Werkzeug nicht einfach und anscheinend noch nicht ausgereift. Es erfordert sogar eine manuelle Umsetzung der Datentypen zwischen VHDL und SystemC. Komfortabler scheinen die Systementwurfswerkzeuge von Transmodeling und Innoveda zu sein, die eine Cosimulation von VHDL, Verilog und SystemC unterstützen.

Es wurde gezeigt, dass die Coverifikation prinzipiell kein Problem darstellt und dass Werkzeuge dafür – wenn auch spärlich – schon vorhanden bzw. in der Entwicklung sind.

5.4 IP-Erstellung und -Nutzung

Eine effektive Methodik für die Wiederverwendung von *Intellectual Property* (IP) erfordert abstrakte IP-Modelle[15]. SystemC scheint sich dafür geradezu aufzudrängen. Die Hersteller von IP werden durch den De-facto-Standard SystemC profitieren, weil sie dadurch nur einen einzigen Satz an Modellen für ihre Abnehmer zur Verfügung stellen müssen. Heute müssen IP-Hersteller ihre Modelle für die verschiedenen C-basierten Umgebungen ihrer Kunden anpassen[16].

Wenn man den empfohlenen Aufbau des Quellcodes mit der Trennung der Moduldeklaration von der Modulfunktionalität und die Separation der Deklarationen in eine eigene Headerdatei beibehält, bietet sich eine einfache Möglichkeit für die Bereitstellung von IP an. Von außerhalb eines Moduls braucht man nur dessen Portnamen und Porttypen kennen, um es in einem Schaltungsentwurf einzusetzen. Es reicht also die Headerdatei und die kompilierte C-Datei für eine geschützte Weitergabe von IP aus. Das Kompilat muss zwar an die Plattform des Kunden angepasst sein, das stellt aber durch die Verwendung von Cross-Compilern kein Problem dar. Der Kunde braucht lediglich die richtigen Bezeichnungen der Ports beim Einbinden des IP-Bausteins zu verwenden und beim Link-Vorgang die bereits kompilierte Objektdatei hinzufügen.

5.5 Synthesefähigkeit

Das derzeit einzige auf dem Markt erhältliche Werkzeug, das die Synthese von SystemC-Code erlaubt, ist der SystemC-Compiler aus der Produktfamilie CoCentric von Synopsys. Ein Baustein aus dem Anwendungsbeispiel wurde nach der Anpassung an die Forderungen des SystemC-Compilers mit diesem erfolgreich synthetisiert. Zur Zeit unterstützt der Compiler nur die Architektursynthese, d.h. RTL-Beschreibungen mit Method-Prozessen können noch nicht bearbeitet werden. Es laufen jedoch Arbeiten, den Compiler in diese Richtung zu erweitern. Überdies ist der Compiler nicht im Stande, direkt auf Gatterebene zu synthetisieren.

Der Codierstil für den SystemC-Compiler unterliegt vielen Einschränkungen, dennoch kann man brauchbare Ergebnisse erzielen. Die Synthese mit diesem Compiler ist sicher noch nicht ausgereift, das kann man sich aber von einer ersten Version nach so kurzem Bestehen der Sprache noch nicht erwarten. An der Weiterentwicklung des SystemC-Compilers wird gearbeitet.

Auch das Unternehmen Frontier arbeitet daran, seine A|RT-Produkte, die bisher C-Modelle für die Systementwicklung verwendet haben, auf SystemC umzustellen. Die

Synthese durch den A|RT-Builder soll dann auch mit SystemC-Code möglich sein.

Generell liegt das Einsatzgebiet von SystemC einer Umfrage zufolge hauptsächlich in der Simulation, Verifikation und Testbench-Erstellung. Nur 2% der Befragten gaben an, SystemC der Synthese wegen zu verwenden. Das liegt wahrscheinlich zu einem großen Teil am fehlenden Vertrauen zu den Werkzeugen für die Verhaltenssynthese oder an den schlechten Erfahrungen, die damit gemacht wurden.

5.6 Vorhandensein von EDA-Werkzeugen mit SystemC - Unterstützung

In Kapitel 4.6.4 wurden Werkzeuge von den Firmen Synopsys, Transmodeling und Innoveda vorgestellt, die die Cosimulation von SystemC mit Hardwarebeschreibungssprachen unterstützen. Darüber hinaus gibt es viele Unternehmen, die an der Entwicklung von entsprechenden Werkzeugen arbeiten.

Napkin to Chip (N2C) ist ein Programmpaket von CoWare, das die Entwicklung von SoC-Entwürfen erleichtern und beschleunigen soll. Die derzeit eingesetzte Modellierungssprache auf Systemebene nennt sich *CowareC* und ist eine Mischung aus ANSI-C und Erweiterungen für die Hardware. Gegen Ende des Jahres soll N2C in der Version 3 auch SystemC-Code verarbeiten können und einen CowareC-SystemC-Übersetzer enthalten.

Die Produktkette A|RT-Designer, A|RT-Builder und A|RT-Library von Frontier erlaubt den Systementwurf in C und die automatische Synthese nach VHDL und Verilog. Auch Frontier ist im Begriff, ihre Bibliotheken um SystemC-Modelle zu erweitern und ihre Werkzeuge auf SystemC umzustellen.

Die Firma Blue Pacific[1] hat ein Werkzeug entwickelt, das eine Cosimulation von SystemC, VHDL und Verilog erlaubt. Allerdings wird nur die SystemC-Version 0.9 unterstützt, in der noch keine Modul-Deklarationen eingeführt worden sind. Ohne die Unterstützung von Modulen ist dieses Werkzeug für den praktischen Einsatz unbrauchbar.

Es ist zu erwarten, dass in den nächsten Monaten bzw. Jahren viele neue SystemC-Produkte auf dem Markt erscheinen werden.

5.7 Durchgängiger Entwurfsablauf

Der vielgepriesene nahtlose SoC-Entwurfsablauf hat noch einige Lücken. Die OSCI-Steering-Group soll die Weiterentwicklung von SystemC in die richtigen Bahnen lenken. In den nächsten Versionen sollen erste Schritte im Software-Entwurfsablauf un-

[1] http://www.bluepc.com

ternommen werden. Software-Module sollen durch ein *Real Time Operating System* (RTOS) synchronisiert werden. Auch die Methoden zur Modellierung auf der UTF-Ebene sind noch nicht vollständig implementiert.

Da die Sprachdefinition einen durchgängigen Ablauf für den Entwurf von Hardware und Software noch nicht erlaubt, ist es nicht verwunderlich, dass auch die entsprechenden Werkzeuge noch nicht existieren. In den vorigen Punkten wurde bereits erwähnt, dass hier viel Aktivität von den Herstellern herrscht.

Die Anwendung von SystemC in einem kommerziellen Projekt wird aufgrund der Stabilität der Sprachdefinition, dem Angebot von Werkzeugen, usw. ein sehr großes Risiko darstellen, zumal zu diesem Zeitpunkt noch kein Chip ausschließlich mit SystemC entwickelt wurde.

Kapitel 6

Zusammenfassung

Der Entwurf von komplexen integrierten Systemen auf Silizium (SoC), die Hardwarekomponenten als auch Software enthalten, muss bei den massiv steigenden Integrationsdichten auf einer höheren funktionellen Ebene beginnen. SystemC ist ein Ansatz, der auf der Sprache C++ basiert und einen nahtlosen Top-Down-Entwurf ermöglichen soll. Im Rahmen der vorliegenden Arbeit wurde untersucht, in wieweit sich SystemC als Grundlage für Entwurf und Verifikation eines typischen Hardware/Software-Systems eignet.

Der Vergleich von SystemC mit VHDL zeigte, dass die Steigerung der Simulationsperformance durch SystemC weit geringer ist als vielfach angepriesen wurde. Als Hauptteil der Arbeit wurde der Systementwurfsablauf anhand eines konkreten Systems mit einem Ethernet-Controller nachvollzogen. Die Beschreibung begann auf abstrakter Ebene zur Verifizierung der Baustein-internen Algorithmen. Nach der Verfeinerung der Schnittstellen wurde die Beschreibung des Ethernet-Controllers so detailliert, dass der Simulation ein hardwarenahes, taktzyklengetreues System zur Verfügung stand. Ein wichtiger Punkt war die Einbindung von SystemC in den bestehenden Entwurfsablauf. Die prinzipielle Machbarkeit der dazu erforderliche Cosimulation mit bestehenden Hardwarebeschreibungssprachen wurde durch die Entwicklung einer Simulationsumgebung bestätigt, welche die Interprozesskommunikationsmechanismen des Betriebssystems sowie das C-Interface des VHDL-Simulators verwendete. Die spärlich vorhandenen Werkzeuge zur Cosimulation wurden kurz vorgestellt. Schließlich wurde die Synthesefähigkeit von SystemC-Code mit dem zur Zeit einzigen auf dem Markt befindlichen Werkzeug, dem SystemC-Compiler von Synopsys, erfolgreich verifiziert. Der Compiler weist aber noch beträchtliche Schwächen auf.

Eine Bewertung der Ergebnisse zeigt, dass SystemC die Anforderungen teilweise erfüllen kann und sich nicht zuletzt, weil es sich lediglich um eine Erweiterung von ANSI C++ handelt, als De-facto-Standard im Bereich IP-Nutzung und System-Level-Spezifikation durchsetzen könnte. Der Grundstock der Sprache SystemC ist definiert.

Etwaige Erweiterungen speziell im Bereich des Software-Entwurfsablaufs werden in den nächsten Monaten erwartet. Viele Firmen unterstützen aktiv bzw. verfolgen passiv die Weiterentwicklung von SystemC.

Das mangelnde Angebot an Werkzeugen, die einen durchgängigen Entwurfsablauf unterstützen, ist ein entscheidender Faktor, warum an einen wirtschaftlichen Einsatz in einem großen Projekt zu diesem Zeitpunkt noch nicht gedacht wird.

Anhang A

Quellcode

A.1 IBUS-Interface-Modell

A.1.1 ibus_if.h

```
#ifndef IBUS_IF_INC
#define IBUS_IF_INC

#include <fstream>
#include "systemc.h"
#include "timer.h"

/* stream for logging */
extern ofstream flog;

#define IBUS_FRAME_SIZE 512   // in Bytes

/* length of ibus_frames in cycles (bits) */
#define INFINITE     -1
#define REQUEST_LEN  46
#define ANSWER_LEN   46
#define RELEASE_LEN  30
#define DATA_LEN     38

enum frame_type {NOTHING, REQUEST, ANSWER, DATA, RELEASE};

enum answer_type {NO_ANSW, ACK, REF_OVL, REF_DMA, REF_INA, REF_BSY};

enum send_state {S_IDLE, S_REQUEST, S_WAIT_FOR_ANSWER,
                 S_ANSWER, S_DATA, S_RELEASE};

enum recieve_state {R_IDLE, R_REQUEST, R_ANSWER,
                    R_WAIT_FOR_DATA, R_DATA, R_WAIT_FOR_RELEASE,
                    R_RELEASE};

struct data_pckt {
   char dest_addr;
   char req_type;
   unsigned char *data;
   int length;

   data_pckt(const char, const char, char *, const int);
   data_pckt();
   friend bool operator ==(const data_pckt&, const data_pckt&);
   friend ostream& operator <<(ostream&, const data_pckt&);
};

struct ibus_frame {
   frame_type type;              // common
   char dest_addr;
   char src_addr;                // for answ-& req-frame
   char req_type;                // for req-frame
   answer_type answ_type;        // for answ-frame
   unsigned char *data;          // for data-frame
   int d_length;                 // length of data
   sc_uint<16> crc;              // for data-frame

   friend bool operator ==(const ibus_frame&, const ibus_frame&);
};

void sc_trace(sc_trace_file *, const data_pckt&, const sc_string&);
void sc_trace(sc_trace_file *, const ibus_frame&, const sc_string&);


SC_MODULE(ibus_if_m){

   /* ports */
   sc_inout<data_pckt> data8_io;
   sc_in<bool> wr_i;
   sc_in<bool> rd_i;
   sc_out<bool> obf_o;
   sc_out<bool> ibf_o;
   sc_in<ibus_frame> ibus_rx_i;
   sc_out<ibus_frame> ibus_tx_o;
   sc_in_clk clk;
   sc_in<bool> reset_i;

   /* signals */
   sc_signal<bool> obf;
   sc_signal<bool> send_t_ex;
   sc_signal<bool> recieve_t_ex;
   sc_signal<int> send_time;
   sc_signal<int> recieve_time;
   sc_signal<int> tx_type;
   sc_signal<bool> send_timer_enable;
   sc_signal<bool> recieve_timer_enable;

   /* variables */
```

```
char address;
data_pckt rx_buffer1, rx_buffer2;
data_pckt tx_buffer;
send_state s_state;
recieve_state r_state;
bool ibf1, ibf2;

/* modules */
timer_m *send_timer;
timer_m *recieve_timer;


SC_CTOR(ibus_if_m){
   SC_THREAD(send);
   sensitive_pos << send_t_ex << obf;
   sensitive << ibus_rx_i << tx_type;

   SC_THREAD(recieve);
   sensitive_pos << recieve_t_ex;
   sensitive << ibus_rx_i << tx_type;

   SC_METHOD(read);
   sensitive_pos << rd_i;

   SC_METHOD(write);
   sensitive_pos << wr_i;

   SC_METHOD(init);
   sensitive_pos << reset_i;

   /* Instantiate */
   send_timer = new timer_m("send_timer");
   send_timer->clk(clk);
   send_timer->enable(send_timer_enable);
   send_timer->din(send_time);
   send_timer->t_ex(send_t_ex);

   recieve_timer = new timer_m("recieve_timer");
   recieve_timer->clk(clk);
   recieve_timer->enable(recieve_timer_enable);
   recieve_timer->din(recieve_time);
   recieve_timer->t_ex(recieve_t_ex);

      /* Initialize */
      send_time = INFINITE;
      recieve_time = INFINITE;
      obf = 0;
      ibf1 = 0;
      ibf2 = 0;
      tx_type = 0;
      send_timer_enable = 1;
      recieve_timer_enable = 1;

   }

   void set_address(const char addr);
   void read();
   void write();
   void send();
   void recieve();
   void init();

   void send_data();
   void get_data(data_pckt&);
   void send_request();
   void send_release();
   void send_answ(const char, answer_type);
   void clear_ibus_tx();

};

#endif
```

A.1.2 ibus_if.cpp

```
#include "systemc.h"
#include "ibus_if.h"

bool operator ==(const data_pckt& d1, const data_pckt& d2){
   /* needed for event() - function of data_pckt */

   if (d1.length != d2.length ||
       d1.dest_addr != d2.dest_addr ||
       d1.req_type != d2.req_type )
       return false;
   for (int i=0; i<d1.length; i++)
```

```
      if (d1.data[i] != d2.data[i])
         return false;
   return true;
}

bool operator ==(const ibus_frame& f1, const ibus_frame& f2){
   /* needed for event() - function of ibus_frame */

   if (f1.type != f2.type ||
       f1.dest_addr != f2.dest_addr ||
       f1.src_addr != f2.src_addr ||
       f1.req_type != f2.req_type ||
       f1.answ_type != f2.answ_type ||
       f1.d_length != f2.d_length)
       return false;
   for (int i=0; i<f1.d_length; i++)
      if (f1.data[i] != f2.data[i])
         return false;
   return true;
}

data_pckt::data_pckt(const char da, const char rt, char *d, const int dl){
   /* Constructor for data_pckt */

   dest_addr = da;
   req_type = rt;
   data = (unsigned char *) d;
   length = dl;
}

data_pckt::data_pckt(){
   /* Constructor for data_pckt */

   dest_addr = 0x00;
   req_type = 0x00;
   data = NULL;
   length = 0;
}

ostream& operator <<(ostream& s, const data_pckt& d){

   s << hex << "data_pckt:  dest_addr = " << int(d.dest_addr) << ", req_type = "
     << int(d.req_type);
   for (int i = 0; i < d.length; i++){
      if (i % 10 == 0)
         s << "\n            ";
      s.width(2);
      s.fill('0');
      s << int(d.data[i]) << ' ';
   }
   return s;
}


void sc_trace(sc_trace_file *tf, const data_pckt& d, const sc_string& NAME){
   /* tracing for data_pckt */

   sc_trace(tf, d.dest_addr, NAME + ".dest_addr");
   sc_trace(tf, d.req_type, NAME + ".req_type");
   sc_trace(tf, d.length, NAME + ".length");
}

void sc_trace(sc_trace_file *tf, const ibus_frame& f, const sc_string& NAME){
   /* tracing for ibus_frame */

   sc_trace(tf, f.type, NAME + ".type");
   sc_trace(tf, f.dest_addr, NAME + ".dest_addr");
   //sc_trace(tf, f.src_addr, NAME + ".src_addr");
   //sc_trace(tf, f.req_type, NAME + ".req_type");
   //sc_trace(tf, f.answ_type, NAME + ".answ_type");
   //sc_trace(tf, f.data, NAME + ".data", bytes_to_trace);
   //sc_trace(tf, f.crc, NAME + ".crc");
}


void ibus_if_m::set_address(const char addr){
   /* set internal address of IBUS-module */

   address = addr;
}

void ibus_if_m::send_request(){
   /* to force REQUEST-frame on ibus_tx_o - port */

   ibus_frame rf;

   rf.type = REQUEST;
   rf.dest_addr = tx_buffer.dest_addr;
```

```
   rf.src_addr = address;
   rf.req_type = tx_buffer.req_type;

   ibus_tx_o = rf;
}

void ibus_if_m::send_answ(const char da, answer_type at){
   /* to force ANSWER-frame on ibus_tx_o - port */

   ibus_frame rf;

   rf.type = ANSWER;
   rf.dest_addr = da;
   rf.src_addr = address;
   rf.answ_type = at;        // has to be changed

   ibus_tx_o = rf;
}

void ibus_if_m::send_release(){
   /* to force RELEASE-frame on ibus_tx_o - port */

   ibus_frame rf;

   rf.type = RELEASE;
   rf.dest_addr = tx_buffer.dest_addr;

   ibus_tx_o = rf;
}

void ibus_if_m::send_data(){
   /* to force DATA-frame on ibus_tx_o - port */

   ibus_frame iframe;

   iframe.type = DATA;
   iframe.dest_addr = tx_buffer.dest_addr;
   iframe.req_type = tx_buffer.req_type;
   iframe.data = tx_buffer.data;
   iframe.d_length = tx_buffer.length;
   iframe.crc = 0;  // has to be extended

   ibus_tx_o = iframe;
}
```

```
void ibus_if_m::clear_ibus_tx(){
   /* to force EMPTY-frame (NOTHING) on ibus_tx_o - port */

   ibus_frame iframe;

   iframe.type = NOTHING;
   iframe.dest_addr = 0;
   iframe.src_addr = 0;
   iframe.req_type = 0;
   iframe.answ_type = NO_ANSW;
   iframe.data = NULL;
   iframe.d_length = 0;
   iframe.crc = 0;

   ibus_tx_o = iframe;
}

void ibus_if_m::get_data(data_pckt& d1){
   /* to read DATA-frame from ibus_rx_i - port */

   ibus_frame iframe = ibus_rx_i;

   d1.dest_addr = iframe.dest_addr;
   d1.req_type = iframe.req_type;
   d1.data = iframe.data;
   d1.length = iframe.d_length;
}

void ibus_if_m::write(){
   /* write data from data8_io into tx_buffer */

   #ifdef LOGGING
      flog << sc_time_stamp()<<": "<<name()<<"::write" << endl;
   #endif
   if (!obf){
      tx_buffer = data8_io;
      obf = 1;
      obf_o = 1;
   }
}

void ibus_if_m::read(){
   /* enables data from rx_buffer (1 or 2) on data8_io - port */
```

```
   #ifdef LOGGING
      flog << sc_time_stamp()<<": "<<name()<<"::read" << endl;
   #endif
   if (ibf1){
      data8_io = rx_buffer1;
      ibf1 = 0;
   }
   else
      if (ibf2){
         data8_io = rx_buffer2;
         ibf2 = 0;
      }
   ibf_o = ibf1 || ibf2;
}

void ibus_if_m::send(){
   /* FSM-process for the send-part of IBUS-module */

   ibus_frame iframe;

   s_state = S_IDLE;
   while (true){
      switch (s_state){
         case S_IDLE:
            #ifdef LOGGING
               flog << sc_time_stamp()<<": "<<name()<<"::send - S_IDLE" << endl;
            #endif
            while (obf.delayed() == 0) wait();
            while (tx_type != NOTHING) wait();
            s_state = S_REQUEST;
            break;

         case S_REQUEST:
            #ifdef LOGGING
               flog << sc_time_stamp()<<": "<<name()<<"::send - S_REQUEST" << endl;
            #endif
            tx_type = REQUEST;
            send_time = REQUEST_LEN;
            send_request();
            do wait(); while (send_t_ex.delayed() == 0);
            clear_ibus_tx();
            send_time = INFINITE;
            tx_type = NOTHING;
            s_state = S_WAIT_FOR_ANSWER;
            break;

         case S_WAIT_FOR_ANSWER:
            #ifdef LOGGING
               flog << sc_time_stamp()<<": "<<name()<<"::send - S_WAIT_FOR_ANSWER" << endl;
            #endif
            do {
               wait();
               iframe = ibus_rx_i;
            } while(iframe.type != ANSWER);
            // !!!!!!
            s_state = S_ANSWER;
            break;

         case S_ANSWER:
            #ifdef LOGGING
               flog << sc_time_stamp()<<": "<<name()<<"::send - S_ANSWER" << endl;
            #endif
            do wait(); while(!ibus_rx_i.event());
            if (iframe.type != ANSWER ||
                iframe.dest_addr != address){
               s_state = S_WAIT_FOR_ANSWER;
            }
            else {
               if (iframe.answ_type == ACK)
                  s_state = S_DATA;
               else
                  s_state = S_RELEASE;
               while (tx_type != NOTHING) wait();
            }
            break;

         case S_RELEASE:
            #ifdef LOGGING
               flog << sc_time_stamp()<<": "<<name()<<"::send - S_RELEASE" << endl;
            #endif
            tx_type = RELEASE;
            send_time = RELEASE_LEN;
            send_release();
            do wait(); while (send_t_ex.delayed() == 0);
            clear_ibus_tx();
            send_time = INFINITE;
            tx_type = NOTHING;
```

```
      s_state = S_IDLE;
      break;

    case S_DATA:
      #ifdef LOGGING
        flog << sc_time_stamp()<<": "<<name()<<":send - S_DATA" << endl;
      #endif
      tx_type = DATA;
      send_time = DATA_LEN+tx_buffer.length*8;
      send_data();
      obf = 0;
      do wait(); while (send_t_ex.delayed() == 0);
      clear_ibus_tx();
      send_time = INFINITE;
      obf_o = 0;
      tx_type = NOTHING;
      s_state = S_IDLE;
      break;

  }
 }
}

void ibus_if_m::recieve(){
  /* FSM-process for the recieve-part of IBUS-module */

  ibus_frame iframe;

  r_state = R_REQUEST;
  while (true) {
    switch (r_state){
      case R_REQUEST:
        iframe = ibus_rx_i;
        if (iframe.type != REQUEST ||
            iframe.dest_addr != address){
          r_state = R_IDLE;
        }
        else {
          #ifdef LOGGING
            flog << sc_time_stamp()<<": "<<name()<<":recieve - R_REQUEST" << endl;
          #endif
          do wait(); while(!ibus_rx_i.event());
          while (tx_type == REQUEST) wait();
          r_state = R_ANSWER;
        }
        break;

      case R_ANSWER:
        #ifdef LOGGING
          flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_ANSWER" << endl;
        #endif
        if (tx_type == DATA)
          send_timer_enable = 0;

        tx_type = ANSWER;
        recieve_time = ANSWER_LEN;
        if (ibf1 && ibf2){
          send_answ(iframe.src_addr, REF_INA); //!!!
          do wait(); while (recieve_t_ex.delayed() == 0);
          r_state = R_WAIT_FOR_RELEASE;
        }
        else {
          send_answ(iframe.src_addr, ACK);
          do wait(); while (recieve_t_ex.delayed() == 0);
          r_state = R_WAIT_FOR_DATA;
        }
        recieve_time = INFINITE;
        if (send_timer_enable){
          clear_ibus_tx();
          tx_type = NOTHING;
        }
        else {
          send_data();
          send_timer_enable = 1;
          tx_type = DATA;
        }
        break;

      case R_WAIT_FOR_DATA:
        #ifdef LOGGING
          flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_WAIT_FOR_DATA" << endl;
        #endif
        do {
          wait();
          iframe = ibus_rx_i;
        } while(iframe.type != DATA);
          // !!!!!!!!!!
```

```
      flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_IDLE" << endl;
      #endif
      do wait(); while(!ibus_rx_i.event());
      r_state = R_REQUEST;
      break;
    }
  }
}

void ibus_if_m::init(){
  /* scans reset-port and initializes the module */

  #ifdef LOGGING
    flog << sc_time_stamp()<<": "<<name()<<"::init" << endl;
  #endif

  clear_ibus_tx();
  ibf_o = 0;
  ibf1 = 0;
  ibf2 = 0;
  obf_o = 0;
  obf = 0;
  send_time = INFINITE;
  recieve_time = INFINITE;
  send_t_ex = 0;
  recieve_t_ex = 0;
  tx_type = NOTHING;
  send_timer_enable = 1;
  recieve_timer_enable = 1;
  s_state = S_IDLE;
  r_state = R_IDLE;
}
```

```
    r_state = R_DATA;
    break;

  case R_DATA:
    #ifdef LOGGING
      flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_DATA" << endl;
    #endif
    if (ibf1 == 0){
      get_data(rx_buffer1);
      ibf1 = 1;
    }
    else {
      get_data(rx_buffer2);
      ibf2 = 1;
    }
    do {
      wait();
      iframe = ibus_rx_i;
    } while(iframe.type == DATA ||
          iframe.type == ANSWER);
    ibf_o = ibf1 || ibf2;
    r_state = R_IDLE;
    break;

  case R_WAIT_FOR_RELEASE:
    #ifdef LOGGING
      flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_WAIT_FOR_RELEASE"
           << endl;
    #endif
    do wait(); while(!ibus_rx_i.event());
    // !!!!!!!!!!!
    r_state = R_RELEASE;
    break;

  case R_RELEASE:
    #ifdef LOGGING
      flog << sc_time_stamp()<<": "<<name()<<"::recieve - R_RELEASE" << endl;
    #endif
    do wait(); while(!ibus_rx_i.event());
    r_state = R_IDLE;
    break;

  case R_IDLE:
    #ifdef LOGGING
```

A.2 Abstrakte Beschreibung des IO-Controllers

A.2.1 io_controller.h

```
#ifndef IO_CONTROLLER_INC
#define IO_CONTROLLER_INC

#include <fstream>
#include "systemc.h"
#include "mbdatm.h"


/* stream for logging */
extern ofstream flog;

SC_MODULE(io_controller_m){

   /* ports */
   sc_inoutmaster<unsigned long, sc_indexed<MEM_SIZE> > i486data_io;
   sc_outmaster<sc_uint<4> > mii_data4_o;
   sc_inslave<sc_uint<4> > mii_data4_i;

   sc_in_clk clk;

   /* variables */
   sc_uint<32> in_fifo[MII_FRAME_SIZE];
   sc_uint<32> out_fifo[MII_FRAME_SIZE];

   unsigned long addr_tx_frame_ptr;
   unsigned long rx_ptr_array;

   /* modules */

   SC_CTOR(io_controller_m){

      SC_SLAVE(control_read, mii_data4_i);

      SC_METHOD(control_write);
      sensitive_pos << clk;

      /* Initialize */
      for (int i = 0; i <  MII_FRAME_SIZE; i++)
         in_fifo[i] = out_fifo[i] = 0;
   }
   void control_write();
   void control_read();

};

#endif
```

A.2.2 io_controller.cpp

```
#include "systemc.h"
#include "io_controller.h"

void io_controller_m::control_write(){

   int word_cnt;

   #ifdef LOGGING
      flog << "TIME: " << sc_time_stamp()<< endl;
   #endif

   #ifdef LOGGING
      flog << "\t\t"<<name()<<": checking for full tx_frame" << endl;
   #endif
   unsigned long tmp = i486data_io[addr_tx_frame_ptr];
   mii_frame *tx_frame_ptr = (mii_frame *) tmp;
   if (tx_frame_ptr != NULL)
      word_cnt = i486data_io[(unsigned long)tx_frame_ptr+(MII_FRAME_SIZE+1)*sizeof(long)];
   // check, if frame available and frame is full (word_cnt == MII_FRAME_SIZE)

   while (tx_frame_ptr != NULL && word_cnt == MII_FRAME_SIZE){

      for (int i = 0; i<MII_FRAME_SIZE; i++){
         // load data into fifo
         out_fifo[i] = i486data_io[(unsigned long)tx_frame_ptr+i*sizeof(long)];
         #ifdef LOGGING
            flog << "\t\t"<< name()<< ": "<<out_fifo[i] << " -> fifo" << endl;
         #endif
      }

      // transmit MII
      for (int i = 0; i < MII_FRAME_SIZE; i++)
```

```
    // multiplex
    for (int g = 0; g < 32; g += 4)
        mii_data4_o = out_fifo[i].range(g+3, g);

    #ifdef LOGGING
        flog << "\t\t"<<name()<<": tx_frame transmitted" << endl;
    #endif

    // write 0xFFFFFFFF (>MII_FRAME_SIZE) into word_cnt
    // to signal software in mbdatm that io-controller has
    // read out the frames and sent successfully
    i486data_io[(unsigned long) tx_frame_ptr+(MII_FRAME_SIZE+1)*sizeof(long)] = 0xFFFFFFFF;

    // read next frame_pointer and word_cnt from MBDATM
    tmp = i486data_io[(unsigned long)tx_frame_ptr+(MII_FRAME_SIZE)*sizeof(long)];
    tx_frame_ptr = (mii_frame *) tmp;
    if (tx_frame_ptr != NULL)
        word_cnt = i486data_io[(unsigned long)tx_frame_ptr+(MII_FRAME_SIZE+1)*sizeof(long)];

  }
}

void io_controller_m::control_read(){

    static sc_uint<32> dword;
    static int index = 0;
    static int fifo_ptr = 0;
    static int arr_ptr = 0;

    sc_uint<4> nibble = mii_data4_i;
    // shift
    dword.range(index+3, index) = nibble;
    index += 4;
    if (index == 32){
        // push data into fifo
        in_fifo[fifo_ptr++] = dword;
        if (fifo_ptr == MII_FRAME_SIZE){
            // fifo full:
            fifo_ptr = 0;

            // read rx_frame_ptr from MBDATM
            #ifdef LOGGING
                flog << "\t\t"<<name()<<": reading rx_frame pointer" << endl;
            #endif

unsigned long rx_frame_ptr = i486data_io[rx_ptr_array+arr_ptr*sizeof(long)];

            if (rx_frame_ptr == 0){
                cerr << "\nIO-Controller has read NULL-ptr from rx_array in MBDATM\n";
                cerr << "MBDATM did not fill rx_array fast enough\n";
                exit(-1);
            }
            if (++arr_ptr == MII_FRAME_SIZE)
                arr_ptr = 0;

            // write data from in_fifo into MBDATM-memory
            for (int i = 0; i < MII_FRAME_SIZE; i++){
                #ifdef LOGGING
                    flog << "\t\t"<<name()<<": " << dword <<" <- fifo" << endl;
                #endif
                i486data_io[rx_frame_ptr + i*sizeof(long)] = in_fifo[i];
            }
            // write 0xFFFFFFFF into word_cnt from frame
            // to indicate the software (MBDATM) that frame has been filled
            #ifdef LOGGING
                flog << "\t\t"<<name()<<": write 0xF..F into rx_frame(word_cnt)" << endl;
            #endif
            i486data_io[rx_frame_ptr + (MII_FRAME_SIZE+1)*sizeof(long)] = 0xFFFFFFFF;
        }
        index = 0;
    }
}
```

A.3 Synthetisierbarer IO-Controller

A.3.1 io_controller.h

```
#ifndef IO_CONTROLLER_INC
#define IO_CONTROLLER_INC

#ifdef LOGGING
#include <fstream>
#endif
#include "systemc.h"

#ifdef LOGGING
/* stream for logging */
extern ofstream flog;
#endif

#define SCAN_INTERVAL 200000   // 200 us
#define NS *1e-9

#define MII_FRAME_SIZE 400

SC_MODULE(io_controller_m){

   /* ports */
   sc_in_clk clk_i486_if;

   sc_out<sc_uint<30> > addr30_o;
   sc_inout<sc_uint<32> > data32_i;
   sc_out<sc_uint<32> > data32_o;
   sc_out<bool> ads_n_o;
   sc_out<bool> wr_n_o;
   sc_in<bool> rdy_n_i;
   sc_in<bool> ar_i;
   sc_in<bool> res_n_i;

   sc_out<sc_uint<4> > mii_data4_o;
   sc_out<bool> mii_en_o;
   sc_in<sc_uint<4> > mii_data4_i;
   sc_in<bool> mii_en_i;
   sc_in<bool> mii_coll_det;
   sc_in_clk clk_mii;

   /* signals */
   sc_signal<bool> start_mux;
   sc_signal<bool> ready_mux;
   sc_signal<bool> start_read;
   sc_signal<bool> out_fifo_reset;

   sc_signal<sc_uint<30> > addr30_cwr;
   sc_signal<sc_uint<32> > data32_o_cwr;
   sc_signal<sc_uint<32> > data32_i_cwr;
   sc_signal<bool> ads_n_cwr;
   sc_signal<bool> wr_n_cwr;
   sc_signal<bool> rdy_n_cwr;
   sc_signal<sc_uint<30> > addr30_crd;
   sc_signal<sc_uint<32> > data32_o_crd;
   sc_signal<sc_uint<32> > data32_i_crd;
   sc_signal<bool> ads_n_crd;
   sc_signal<bool> wr_n_crd;
   sc_signal<bool> rdy_n_crd;

   /* variables */
   sc_uint<32> addr_tx_frame_ptr;
   sc_uint<32> rx_ptr_array;
   sc_uint<32> shared_mem1[MII_FRAME_SIZE];  // for write
   sc_uint<32> shared_mem2[MII_FRAME_SIZE];  // for read

   SC_CTOR(io_controller_m){

      SC_CTHREAD(control_write, clk_i486_if.pos());
      //watching(mii_coll_det.delayed() == true);
      watching(res_n_i.delayed() == false);

      SC_CTHREAD(control_read, clk_i486_if.pos());
      watching(res_n_i.delayed() == false);

      SC_CTHREAD(mux, clk_mii.pos());
      watching(res_n_i.delayed() == false);
      SC_CTHREAD(shift, clk_mii.pos());
      watching(res_n_i.delayed() == false);

      SC_CTHREAD(arbiter,clk_i486_if.pos());
      watching(res_n_i.delayed() == false);

      /* Initialize */
      start_mux = 0;
```

```
      ready_mux = 0;
      start_read = 0;
      out_fifo_reset = 0;

      addr30_cwr = sc_uint<30>(0);
      data32_o_cwr = sc_uint<32>(0);
      data32_i_cwr = sc_uint<32>(0);
      ads_n_cwr = 1;
      wr_n_cwr = 1;
      rdy_n_cwr = 1;
      addr30_crd = sc_uint<30>(0);
      data32_o_crd = sc_uint<32>(0);
      data32_i_crd = sc_uint<32>(0);
      ads_n_crd = 1;
      wr_n_crd = 1;
      rdy_n_crd = 1;


      // init shared memory
      for (int i=0; i < MII_FRAME_SIZE; i++)
         shared_mem1[i] = shared_mem2[i] = 0;
   }
   void control_write();
   void control_read();
   void mux();
   void shift();

   sc_uint<32> read_from_memory(sc_uint<32>);
   void write_into_memory(sc_uint<32>, sc_uint<32>);

   sc_uint<32> read_from_memory_cwr(sc_uint<32>);
   void write_into_memory_cwr(sc_uint<32>, sc_uint<32>);
   sc_uint<32> read_from_memory_crd(sc_uint<32>);
   void write_into_memory_crd(sc_uint<32>, sc_uint<32>);
   void arbiter();
};

#endif
```

A.3.2 io_controller.cpp

```
#include "systemc.h"
#include "io_controller.h"
```

```
void io_controller_m::mux(){

   /* synopsys resource Shr_read_ram:
     variables = "shared_mem1",
     map_to_module = "RR_S_2P_1024_8_wrap",
     memory_address_ports = "adr0"; */
  sc_uint<32> data;

   while (true){
      while (start_mux.read() == 0) wait();
      #ifdef LOGGING
         flog << sc_time_stamp()<<": "<<name()<<"::select - enabled" << endl;
      #endif
      mii_en_o = 1;
      wait(); //rocco
      for (int i = 0; i < MII_FRAME_SIZE; i++){
         data = shared_mem1[i];

         mii_data4_o = data.range(3,0);
         wait();
         mii_data4_o = data.range(7,4);
         wait();
         mii_data4_o = data.range(11,8);
         wait();
         mii_data4_o = data.range(15,12);
         wait();
         mii_data4_o = data.range(19,16);
         wait();
         mii_data4_o = data.range(23,20);
         wait();
         mii_data4_o = data.range(27,24);
         wait();
         mii_data4_o = data.range(31,28);
         wait();
      }
      mii_en_o = 0;
      mii_data4_o = 0;
      ready_mux = 1;
      wait();
      ready_mux = 0;
      wait();
   }
}
```

```
void io_controller_m::shift(){

    /* synopsys resource Shr_read_ram:
       variables = "shared_mem2",
       map_to_module = "RR_S_2P_1024_8_wrap",
       memory_address_ports = "adr1"; */
    sc_uint<32> data;

    while (true){
      while (mii_en_i.read() == false) wait();
      #ifdef LOGGING
        flog << sc_time_stamp()<<": "<<name()<<"::collect - enabled" << endl;
      #endif

      for (int i = 0; i < MII_FRAME_SIZE; i++){
          data.range(3,0) = mii_data4_i;
          wait();
          data.range(7,4) = mii_data4_i;
          wait();
          data.range(11,8) = mii_data4_i;
          wait();
          data.range(15,12) = mii_data4_i;
          wait();
          data.range(19,16) = mii_data4_i;
          wait();
          data.range(23,20) = mii_data4_i;
          wait();
          data.range(27,24) = mii_data4_i;
          wait();
          data.range(31,28) = mii_data4_i;
          shared_mem2[i] = data;
          wait();
      }
      start_read = 1;
      wait();
      start_read = 0;
      wait();
    }
  }

  // arbiter functions ----------------------------------------
  sc_uint<32> io_controller_m::read_from_memory(sc_uint<32> mp){
      // read from mbdatm-memory over i486-IF

      addr30_o = mp;
      ads_n_o = 0;
      wr_n_o = 0;
      wait();
      ads_n_o = 1;
      while (rdy_n_i == 1) wait();
      sc_uint<32> data = data32_i.read();
      wr_n_o = 1;
      addr30_o = 0;
      return data;
  }

  void io_controller_m::write_into_memory(sc_uint<32> mp,  sc_uint<32> data){

      addr30_o = mp;
      ads_n_o = 0;
      wr_n_o = 1;
      wait();
      ads_n_o = 1;
      data32_o = data;
      while (rdy_n_i == 1) wait();
      wr_n_o = 1;
      addr30_o = 0;
      data32_o = 0;
  }

  // control_write functions ----------------------------------
  sc_uint<32> io_controller_m::read_from_memory_cwr(sc_uint<32> mp){

      // read from mbdatm-memory over i486-IF

      addr30_cwr = mp >> 2;
      ads_n_cwr = 0;
      wr_n_cwr = 0;
      wait(); // rocco
      while (rdy_n_cwr == 1) wait();
      sc_uint<32> data = data32_i_cwr.read();
      wr_n_cwr = 1;
      addr30_cwr = sc_uint<30>(0);
      ads_n_cwr = 1;
      return data;
  }
```

```
void io_controller_m::write_into_memory_cwr(sc_uint<32> mp, sc_uint<32> data){

    addr30_cwr = mp >> 2;
    ads_n_cwr = 0;
    wr_n_cwr = 1;
    data32_o_cwr = data;
    wait(); // rocco
    while (rdy_n_cwr == 1) wait();
    wr_n_cwr = 1;
    addr30_cwr = sc_uint<30>(0);
    ads_n_cwr = 1;
    data32_o_cwr = sc_uint<32>(0);
}

// control_read functions ----------------------------------
sc_uint<32> io_controller_m::read_from_memory_crd(sc_uint<32> mp){

    // read from mbdatm-memory over i486-IF

    addr30_crd = mp >> 2;
    ads_n_crd = 0;
    wr_n_crd = 0;
    while (rdy_n_crd == 1) wait();
    sc_uint<32> data = data32_i_crd.read();
    wr_n_crd = 1;
    addr30_crd = sc_uint<30>(0);
    ads_n_crd = 1;
    return data;
}

void io_controller_m::write_into_memory_crd(sc_uint<32> mp, sc_uint<32> data){

    addr30_crd = mp >> 2;
    ads_n_crd = 0;
    wr_n_crd = 1;
    data32_o_crd = data;
    while (rdy_n_crd == 1) wait();
    wr_n_crd = 1;
    addr30_crd = sc_uint<30>(0);
    ads_n_crd = 1;
    data32_o_crd = sc_uint<32>(0);
}
```

```
void io_controller_m::control_write(){
    /* synopsys resource Shr_read_ram:
       variables = "shared_mem1",
       map_to_module = "RR_S_2P_1024_8_wrap",
       memory_address_ports = "adr1"; */
    sc_uint<32> word_cnt;

    while (res_n_i.read() == 0) wait();

    // initialize

    // wait for 1. AR (HWS-Daten)
    while (ar_i.read() == 0) wait();
    sc_uint<32> hws = data32_i.read();

    wait();

    // wait for 2. AR (ACB-Pointer)
    while (ar_i.read() == 0) wait();
    addr_tx_frame_ptr = data32_i.read();
    wait();

    while(true){
      // normally Attention Request - Signal from MBDATM
      // would wake up IO-Controller to read data from the memory,
      // but the model from Hr. Wahl said: wait for some ms !!!

      wait(10000);

      #ifdef LOGGING
        flog << sc_time_stamp()<<": "<<name()<<"::control_write - Attention Request"
             << endl;
      #endif

      sc_uint<32> tx_frame_ptr = read_from_memory_cwr(addr_tx_frame_ptr);
      wait();
      if (tx_frame_ptr != 0)
        word_cnt = read_from_memory_cwr(tx_frame_ptr+(MII_FRAME_SIZE+1)*4);
      else // rocco
        wait();
      // check, if frame available and frame is full (word_cnt == MII_FRAME_SIZE)
      while (tx_frame_ptr != 0 && word_cnt == MII_FRAME_SIZE){
        #ifdef LOGGING
```

```
            flog << sc_time_stamp()<<": "<<name()
                 <<"::control_write - writing mii_frame into out_fifo" << endl;
         #endif

         for (int i = 0; i<MII_FRAME_SIZE; i++){
            // reading from i486-IF and writing into
            // out_fifo is mixed, so read_from_memory_cwr could not be applied
            sc_uint<32> data = read_from_memory_cwr(tx_frame_ptr+i*4);

            if (i == 0){
               start_mux = 1;
               shared_mem1[i] = data;
               wait();
               start_mux = 0;
            }
            else {
               shared_mem1[i] = data;
               wait();
            }
            // wait(); ??
         }
         while (ready_mux == 0) wait();

         // write 0xFFFFFFFF (>MII_FRAME_SIZE) into tx_frame_ptr
         // to signal software in mbdatm that io-controller has
         // read out the frames and sent successfully
         write_into_memory_cwr(tx_frame_ptr+(MII_FRAME_SIZE+1)*4, 0xFFFFFFFF);
         wait();

         // read next frame_pointer and word_cnt from MBDATM
         tx_frame_ptr = read_from_memory_cwr(tx_frame_ptr+MII_FRAME_SIZE*4);
         wait();
         if (tx_frame_ptr != 0)
            word_cnt = read_from_memory_cwr(tx_frame_ptr+(MII_FRAME_SIZE+1)*4);
         else // rocco
           wait();
      }

   }
}

void io_controller_m::control_read(){

  /* synopsys resource Shr_read_ram:
       variables = "shared_mem2",
       map_to_module = "RR_S_2P_1024_8_wrap",
       memory_address_ports = "adr0"; */
   int arr_ptr = 0;

   while (true){
      while (start_read.read() == 0) wait();
      #ifdef LOGGING
         flog << sc_time_stamp()<<": "<<name()<<"::control_read " << endl;
      #endif

      // read rx_frame_ptr from MBDATM
      sc_uint<32> rx_frame_ptr = read_from_memory_crd(rx_ptr_array+arr_ptr*4);
      wait();

      if (++arr_ptr == MII_FRAME_SIZE)
         arr_ptr = 0;

      // write data from in_fifo into MBDATM-memory
      for (int i = 0; i < MII_FRAME_SIZE; i++){
         sc_uint<32> d = shared_mem2[i];
         write_into_memory_crd(rx_frame_ptr + i*4, d);
         wait();

      }

      // write 0xFFFFFFFF into word_cnt from frame
      // to indicate the software (MBDATM) that frame has been filled
      write_into_memory_crd(rx_frame_ptr + (MII_FRAME_SIZE+1)*4, 0xFFFFFFFF);
      wait();
   }
}

void io_controller_m::arbiter(){

   sc_uint<30> addr;
   sc_uint<32> data;


   while(true){
      // control_write gets control over i486-iF
      if (ads_n_cwr == 0) {
         addr = addr30_cwr.read();
         if (wr_n_cwr){
```

```
    data = data32_o_cwr.read();
    write_into_memory(addr, data);
      rdy_n_cwr = 0;
      wait();
    }
    else {
      // read data from memory
      data = read_from_memory(addr);
      data32_i_cwr = data;
      rdy_n_cwr = 0;
      wait();
    }
    rdy_n_cwr = 1;
  }
  // control_read gets control over i486-iF
  if (ads_n_crd == 0) {
    addr = addr30_crd.read();
    if (wr_n_crd){
      data = data32_o_crd.read();
      write_into_memory(addr, data);
        rdy_n_crd = 0;
        wait();
      }
      else {
        // read data from memory
        data = read_from_memory(addr);
        data32_i_crd = data;
        rdy_n_crd = 0;
        wait();
      }
      rdy_n_crd = 1;
    }
    wait();
  }
}
```

Abkürzungen

ACB	ATM Control Block
ALU	Arithmetic Logic Unit
ANSI	American National Standard Institute
AR	Attention Request
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
ATMC	ATM Controller
BCA	Bus Cycle Accurate
BC	Behavioral Compiler
CA	Cycle Accurate
DAC	Design Automation Conference
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FIFO	First In First Out
FIR	Finite Impulse Response
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GB	Giga Byte
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
IEEE	Institude of Electrical and Electronics Engineers
IF	Interface
IO	Input/Output
IOC	Input/Output Controller
IP	Intellectual Property
IPC	Inter Process Communication
ISDB	Integrated Signal Data Base

MB	Mega Byte
MII	Media Independent Interface
OOP	Object Oriented Programming
OSCI	Open SystemC Initiative
PMD	Physical Media Dependent
RAM	Random Access Memory
ROM	Read Only Memory
RPC	Remote Procedure Call
RTL	Register Transfer Level
RTOS	Real Time Operating System
SLD	System Level Design
SoC	System On Chip
SW	Software
TF	Timed Functional
UTF	Untimed Functional
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
WIF	Waveform Intermediate Format

Index

Literaturverzeichnis

[1] S. Liao, S. Tjiang, and R. Gupta. An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment. *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 70–75, 1997.

[2] J. Notbauer, T. Albrecht, and S. Rohringer. Verification and Management of a multi-million gate embedded core design. *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 425–428, 1999.

[3] J. Notbauer, T. Albrecht, and G. Niedrist. HW/SW Coverification: Challenges and Experiences in a Real-World Design. *DATE Conference Paris*, 2000.

[4] L. Semeria and A. Ghosh. Methodology for Hardware/Software Coverification in C/C++. Technical report, http://www.systemc.org/papers/05b_4.pdf, 2000.

[5] T. Albrecht, J. Notbauer, and S. Rohringer. HW/SW Coverification Performance Estimation & Benchmark for a 24 RISC Core Design. *35th ACM/IEEE Design Automation Conference*, pages 808—811, 1998.

[6] D. Dietrich. *Skriptum zur Vorlesung: Komplexe Schaltwerke und ASIC-Entwicklung*. Institut für Computertechnik, TU Wien, 1997.

[7] B. Bailey, R. Klein, and S. Leef. Hardware/Software Co-Simulation Strategies for the Future. Technical report, http://www.mentor.com/seamless, 2000.

[8] C. Ajluni. System-Level Languages Fight To Take Over As The Next Design Solution. Technical report, http://www.elecdesign.com, Juni 2000.

[9] K. Riedling. *Technisches Programmieren in C++*. Institut für Angewandte Elektronik und Quantenelektronik, TU Wien, fourth edition, 1999.

[10] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000.

[11] Synopsys, Coware, and Frontier. *SystemC User's Guide Version 1.1*. http://www.systemc.org, 2000.

[12] H. Muhr, G. Cadek, J. Notbauer, and G. Niedrist. Einsatz von C-basierten Methoden in der Systementwicklung. *Austrochip*, 2000.

[13] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice-Hall, third edition, 1998.

[14] Synopsys. *SystemC Compiler Workshop*, 2000.

[15] M. Grant. Design Methodologies for System Level IP. Technical report, Cadence Design Systems, Alta Business Unit, 1999.

[16] K. Bartleson. A New Standard for System-Level Design. Technical report, http://www.systemc.org, September 1999.