SoC System Level Integration

Master's Thesis

by

Karsten Larsen

February 25, 2003

Supervisors: Jan Madsen & Jens Sparsø

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark

Preface

This thesis has been prepared at the department of Computer Science and Engineering at institute of Informatics and Mathematical Modelling, Technical University of Denmark in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

I wish to thank Ph.D. Özgün Paker for his indispensable assistance regarding the comprehension of the mini-core system. Furthermore I wish to thank fellow students Nicolai Jørgensen and Andreas Lorentzen for sharing their SystemC experience with me, and Anne Marie Valentin Hansen for her assistance in the creation of this report.

Lyngby, Denmark
February 25th, 2003
Karsten Larsen

Abstract

The trend within SoC (System-on-Chip) design is a decreased time-to-market. One of the measures to comply with this is to reuse previously designed cores or to purchase IP (Intellectual Property) cores, thus decreasing design time. In order easily to employ such cores in a design, the cores must be constructed independently of specific communication formats, i.e. the cores must utilize general purpose communication interfaces suitable for a variety of designs.

To adapt the cores to the communication format of the SoC on-chip interconnect, an external module must be constructed converting between the communication format of the core and of the interconnect. Such an architecture is often referred to as *communication wrapping*. In this thesis general purpose interfaces are defined and communication wrappers are constructed for both cores and interconnect, allowing easy substitution of both. The communication format employed between wrapped cores and wrapped interconnect is the OCP (Open Core Protocol). OCP is what often is referred to as a *core-centric* protocol, indicating that it is designed with emphasis on the needs of the core. Furthermore constructed are methods capable of logging information about the data traffic of the SoC. The information is stored in files allowing post processing of the data.

The communication format of OCP wrapped cores and interconnect including logging of data traffic is introduced in an existing SoC. This SoC is a heterogeneous processor architecture developed specifically for flexible low power digital signal processing. The system referred to as the mini-core system currently contains two core types, both processors dedicated at a specific digital signal processing task.

The entire mini-core system has been reimplemented as a behavioral model to simplify the design, in order to ease experimenting with and testing the system. All implementation has been performed in SystemC.

The behavioral mini-core system has been tested with the original communication format and with the OCP wrapping architecture. The test showed unaltered functionality. The advantages and disadvantages of the implemented structure are assessed and discussed.

Resumé

Udviklingen indenfor SoC (System-on-Chip) design går mod kortere design- og udviklingstid. En af mulighederne for at imødekomme dette krav er at genbruge tidligere designede *cores* eller at købe *IP* (*Intellectual Property*) *cores*, for på denne måde at formindske design tiden. For at være i stand til at benytte sådanne *cores* i et design, må de konstrueres uafhængigt a specifikke kommunikationsformater, de må altså benytte sig af et *general purpose* kommunikations-*interface* anvendeligt i forskellige design.

For at tilpasse *cores* til kommunikationsformatet anvendt af SoC's *on-chip interconnect* er det nødvendigt at konstruere et eksternt modul til at konvertere mellem kommunikationsformaterne benyttet af *core* og *interconnect*. En sådan struktur bliver ofte refereret til som kommunikations-*wrapping* (indhylling). I dette projekt defineres og konstrueres *general purpose interfaces* og kommunikations-*wrappere* for både *cores* og *interconnect*, hvilket tillader let udskiftning af begge. Det benyttede kommunikationsformat mellem *cores* og *interconnect* er OCP (Open Core Protocol). OCP er, hvad der ofte refereres til som en *core-centric* protokol, hvilket betyder, at protokollen tager udgangspunkt i *corens* behov. Ydermere er der konstrueret funktioner i stand til at opsamle information om data trafikken i en SoC. Denne information bliver lagret i filer, hvilket tillader post processering af den opsamlede data.

Kommunikationsformatet inkluderende OCP-*wrapped cores* og *interconnect* bliver introduceret i et allerede eksisterende SoC. Dette SoC er en heterogen processor arkitektur udviklet specielt til lav-effekt digital signalbehandling. Systemet, der refereres til som mini-core systemet, indeholder to *core* typer, begge processorer dedikeret til en specifik digital signalbehandling opgave.

Hele mini-core systemet er blevet reimplementeret som en *behavioral* model for at simplificere designet. Dette letter test af og eksperimentering med systemet. Al implementering er foretaget i SystemC.

Det *behavioral* mini-core system er blevet testet med det original kommunikationsformat og med OCP-*wrapper* strukturen. Testen viser, at funktionaliteten er uændret. Fordelene og ulemperne ved den implementerede struktur bliver vurderet og diskuteret.

Abbreviations

- ALU - Arithmetic-Logic Unit.
- AMBA - Advanced Microprocessor Bus Architecture.
- API - Application Programming Interface.
- CC - Clock Cycle.
- CPU - Central Processing Unit.
- DSP - Digital Signal Processor/Processing.
- FIR - Finite Impulse Response.
- FPGA - Field Programmable Gate Array.
- GNU - GNU's Not Unix.
- GPL - General Public License.
- HDL - Hardware Description Language.
- I/O - Input/Output.
- IC - Integrated Circuit.
- IEEE - Institute of Electrical and Electronics Engineers.
- IIR - Infinite Impulse Response.
- IP - Intellectual Property.
- JTAG - Joint Test Action Group.
- LGPL - Lesser General Public License.
- OCP - Open Core Protocol.
- OCP-IP - Open Core Protocol - International Partnership.
- PC - Program Counter.

- RTL - Register Transfer Level.
- SoC - System-on-Chip.
- VHDL - Very high speed integrated circuit Hardware Description Language.
- VSIA - Virtual Socket Interface Alliance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the motivation and objectives for this thesis. Furthermore the organization of the report is described.

1.1 Motivation

For almost 40 years[1] the well known *Moore's Law*, stating that the logical density of integrated circuits (IC) doubles every 18 months, has been accepted as a fact. At this certain point of time however the *Semiconductor Industry Association*, formerly claiming this to continue until at least 2015, are now stating that this will not continue beyond 2007 - unless a drastic progress within semiconductor fabrication is made before that time [18].

However, SoCs (System-on-Chip) have all ready now reached a very high degree of complexity. Furthermore although one assumes the progress to slow down, the fact that the time-to-market for a design is decreasing cannot be disregarded. Table 1.1 shows the progress within SoC designs the last few years.

	1997	**1998**	**1999**	**2002**	**Delta**
Process Technology	0.35μ	0.25μ	0.18μ	0.13μ	$\sim$7x
Gate Count	200-500K	1-2M	4-6M	10-25M	$\sim$50x
Design Cycle*	12-18	10-12	8-10	6-8	$\sim$2x
Derivative Design Cycle*	6-8	4-6	2-4	2-3	$\sim$2x

Table 1.1: SoC design tendencies from 1997 to 2002 [3]. **Derivative Design Cycle** refers to the implementation period of the design. Notice that the **Delta** field of **Process Technology** shows the approximate reduction of *area*. * In months.

Even though there is uncertainty regarding the progress within SoC design the following years, most likely the requirements toward SoC designers

[1] First published by Gordon Moore in 1965

will continue to increase.

To fulfill these requirements of increased productivity, designers are realizing that it is necessary to design the elements of a system concurrently. This is achieved by dividing the design into clearly defined blocks (referred to as *cores*). With these cores defined it is reasonable to examine, if it is somehow possible to avoid designing some of them thus saving time. This can be achieved either by reusing cores from old designs or by purchasing IP (Intellectual Property) cores available from various vendors.

To reuse or purchase cores it is necessary to construct the cores independent of specific communication structures. The cores must therefore be equipped with a general purpose communication interface, which the designers must adapt to the current design. Such core interfaces are often referred to as *core-centric*.

1.2 Objective

The objective of this project is to develop a methodology resulting in easy IP core integration at system level, requiring the definition of a general purpose core interface. Several core interface standards have been introduced for the SoC market to ease the integration of cores in SoCs. This thesis examines a few of these standards, and based hereupon a core interface is defined.

Based on the same principles as the core interface a general purpose interconnect interface is defined, allowing easy interchanging of interconnect format (bus, ring etc.). A bus interconnect module compliant with this interface is constructed to illustrate and test the principles employed.

Furthermore methods for extracting traffic information from a SoC are constructed. Together with the easy integration of cores and various interconnect formats, this results in easy benchmarking of the communication profile of a SoC for various interconnect formats.

The interfaces described above are introduced in an existing SoC system to validate the functionality. The system is a heterogeneous processor architecture aimed at low-power DSP (Digital Signal Processing). The cores of the system are processors dedicated at performing a specific DSP task, for instance FIR (Finite Impulse Response) and IIR (Infinite Impulse Response) filtering. This dedication results in compact cores with small instruction sets optimized for the specific task. Because of this the cores are referred to as mini-cores, with the entire system referred to as the mini-core system.

The mini-core system was originally implemented in VHDL (Very high speed integrated circuit Hardware Description Language) and synthesized into an ASIC (Application Specific Integrated Circuit). The system is reimplemented as a behavioral model in SystemC, thus allowing easy modifica-

tions of the communication method due to the increased abstraction level.

All implementation is performed in SystemC version 2.0 [19] and [20].

1.3 Organization of the Thesis

Chapter 2 "SoC Communication" provides an introduction to some of the communication methods employed in SoCs. Furthermore a short description of some of the most commonly used communication protocols for SoCs is provided.

Chapter 3 "OCP Protocol" provides a more detailed description of the OCP (Open Core Protocol) concerning an implementation in the mini-core system.

Chapter 4 "Mini-Core System" is a description of the mini-core system serving as basis for a SystemC behavioral implementation.

Chapter 5 "Implementation of Mini-Cores" describes the SystemC implementation of the FIR and IIR processor mini-cores based on a discussion of HDLs (Hardware Description Languages) and processor architectures. Furthermore is described the construction of 2 additional mini-cores designed to aid in testing of the system.

Chapter 6 "Implementation of Mini-Core System Communication" is a description of the SystemC implementation of communication modules for the mini-core system . This include the original communication format, an OCP core interface, an OCP interconnect interface, a behavioral bus interconnect compliant with the OCP interconnect interface and methods for logging data traffic information.

Chapter 7 "Test of Mini-Core System Implementation" presents the test of the behavioral SystemC mini-core system implemented in this thesis.

Chapter 8 "Discussion" discusses the results achieved in this thesis. The possibilities of the constructed mini-core system are described, as well as the advantages and disadvantages of employing the communication methods described in chapter 6.

Chapter 9 "Conclusion" summarizes and concludes the thesis.

Chapter 2

SoC Communication

This chapter provides a brief introduction to some of the communication methods currently employed in SoCs designs, aimed at later implementation in the mini-core system. First is described the difference between *core-centric* and *bus-centric* communication protocols and the progress within this area. Following is a description of how *communication wrappers* are employed in SoCs. Finally is a short description of commonly used SoC communication protocols and a comparison of these regarding this project.

2.1 Core-Centric vs. Bus-Centric Communication Protocols

In the past is has been customary to construct SoCs with the communication of the cores aimed at a specific industrial network format - most often a bus, e.g. the AMBA (Advanced Microprocessor Bus Architecture) bus [1]. This is what is referred to as a *bus-centric* protocol. Currently the tendency is to wish for high portability of SoC cores, thus being able to reuse or purchase existing cores saving a great amount of design time. Bus-centric designed cores are most often difficult to port to other network types because the communication format normally is incorporated in the cores.

To increase the portability of SoC cores the concept of *core-centric* protocols has been been suggested. The basic idea is to equip the cores with a general purpose interface. Designers implementing such cores in a design, must incorporate a *bridge* between the cores and the communication network of choice. This can in fact be described as adding an extra (more abstract) *communication layer*. Figure 2.1 shows the difference between a core equipped with a core-centric and a bus-centric interface.

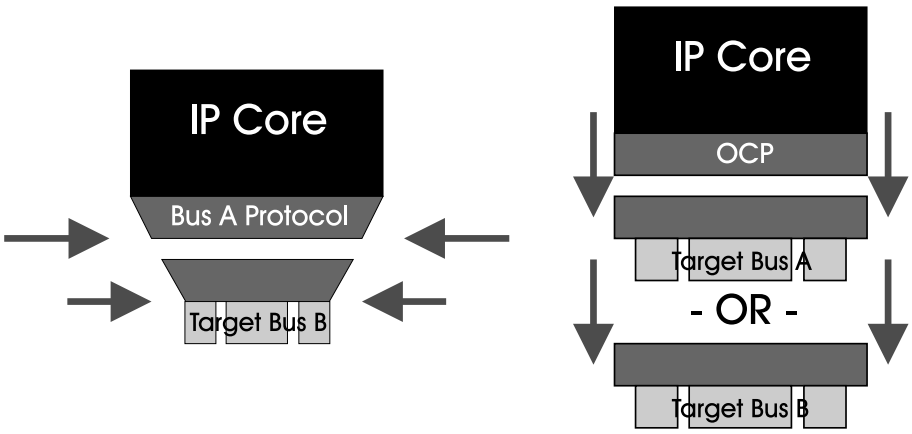

Figure 2.1: Comparison of bus-centric and core-centric communication protocol [12]. Left: Core with bus-centric wrapper suitable only for a specific target bus. Right: Core with core-centric wrapper enabling the core to function with several different target buses. OCP [11] is a specific core-centric protocol.

2.2 Communication Wrapping

The concept of *communication wrapping* is extremely useful regarding SoC communication and is much employed in the industry. Communication wrapping indicates removing all protocol specific communication (being it core-centric or bus-centric) from the cores, placing it in external modules. The cores will interact with the surroundings by simple primitives such as *send* and *receive*. The external module can be thought of as *wrapping* the core, hence the expression. This introduces an extra, more abstract interface between the core and wrapper and hence also an extra communication layer. Figure 2.2 shows a core with a core-centric interface also equipped with a communication wrapper adding an extra communication layer.

When porting a wrapped core only the wrapper needs to be altered not the core itself. Furthermore by implementing the core/wrapper interface with simple primitives, it should be possible to reuse the interface for several different core types.

Another advantage is that the manufacturer of a core might equip it with a variety of different communication wrappers complying with a range of commonly used communication protocols. This way designers can acquire the core together with a wrapper suitable for the design. If changing the communications format, a new wrapper can be acquired reusing the core itself.

It is also possible to wrap networks, thus changing the communication format, as seen in figure 2.3. This can perhaps increase the abstraction level

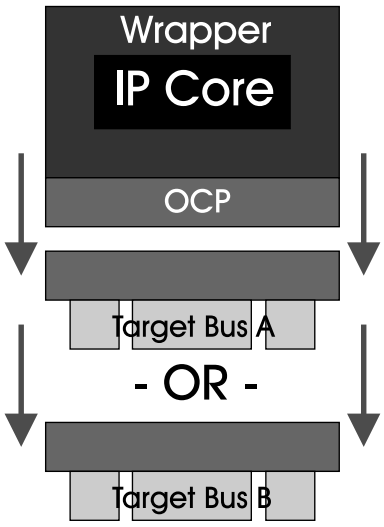

Figure 2.2: Core-centric design with communication wrapper. Based on figure 2.1 [12]

.

resulting in a simpler communications format seen from the cores, or it can add features not available in the original network. Furthermore one could imagine network manufacturers constructing wrappers for their networks, resulting in a more abstract communications protocol.

If a standard was introduced in this area it could result in a *plug-and-play* like behavior, where it would be possible to acquire (or construct) a core with a wrapper for a standard protocol along with a network also with a wrapper for this standard. The next section will introduce a few communication protocols which all could develop into standards.

One major disadvantage of employing communication wrapping is the possibility of introducing a logic overhead, which can result in lower speed, larger die area and/or lower power efficiency. An evaluation has to be performed for the actual design to estimate the introduced overhead, and to decide if it is acceptable.

2.3 SoC Protocols

The market of SoC interconnection formats is quite confusing. The reason is that the manufacturers have quite different approaches to the architecture of the interconnects, making it difficult to compare them. This section is an attempt to examine the most commonly employed SoC protocols aimed at choosing one of these for implementation. [17] contains a list of what is referred to as *SoC interconnection buses*. The list includes 13 different

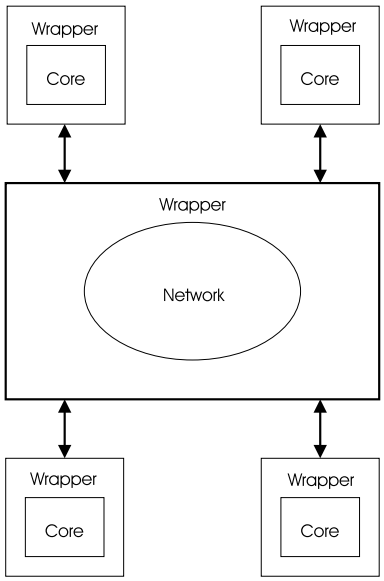

Figure 2.3: Communications wrapping of network.

interconnection brands. Of these the most commonly used are:

- AMBA, ARM [1].
- CoreConnect, IBM [8].
- OCP, International Partnership (OCP-IP) [11].
- VSIA On-chip Bus, Virtual Socket Interface Alliance [21].
- WISHBONE Interconnect, Silicore Corporation [22].

The term *interconnection buses* used in [17] is a somewhat misleading term. Of the interconnect formats mentioned above, AMBA and CoreConnect are best described as buses. This is however not the case for OCP, VSIA and WISHBONE. VSIA is the definition of a network independent core interface, sometimes referred to as a *socket*. OCP is a superset of the VSIA adding additional features for testing etc. WISHBONE is in many ways similar to OCP.

It is quite reasonable to declare AMBA and CoreConnect as being bus-centric communication protocols, and to declare VSIA, OCP and WISHBONE as being core-centric communication protocols. This of course makes VSIA, OCP and WISHBONE the most interesting protocols for this thesis, due to their greater portability. As OCP is a superset of VSIA, VSIA has been disregarded in this thesis. OCP and WISHBONE are chosen as the

most interesting SoC communication protocols for this thesis of the protocols examined. The following section provides a short introduction to the two protocols.

2.4 Introduction to OCP and WISHBONE

Both OCP and WISHBONE define a standardized general purpose interface for the implementation of cores in SoC, and both employ a MASTER/SLAVE architecture. A core that is to initiate communication must do so through a MASTER. The MASTER initiates a core-to-core communication cycle by posting a request and a transaction type. The transaction types are quite simple such as READ and WRITE, OCP however also has more advanced types such as BURST. If the MASTER expects a response (depending on transaction type and configuration), it waits for a such.

2.4.1 Interconnect Format

OCP only support point-to-point communication between MASTER and SLAVE. This structure is seen in figure 2.4. It is necessary to introduce a secondary type of interconnect to transfer data between the cores. OCP is indifferent of the interconnect format, as long as there is supplied wrappers for the interconnect. In figure 2.4 the interconnect is shown as a bus but it could just as well be a ring, hypercube etc.

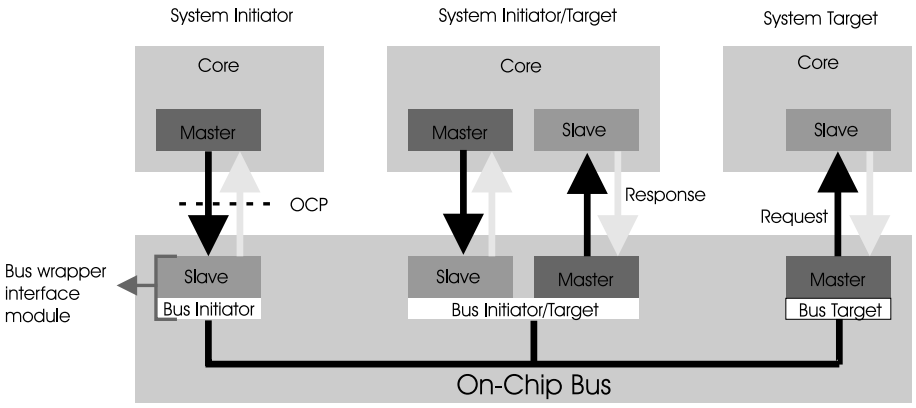

Figure 2.4: Point-to-point MASTER/SLAVE architecture of OCP[11].

WISHBONE supports 4 types of interconnect formats between MASTER and SLAVE: Point-to-point, data-flow (pipeline), shared bus and crossbar. These interconnects are seen in figure 2.5. This is basically achieved by adding enable signals to the WISHBONE interface. The interconnect (such as the crossbar) is itself constructed as a core.

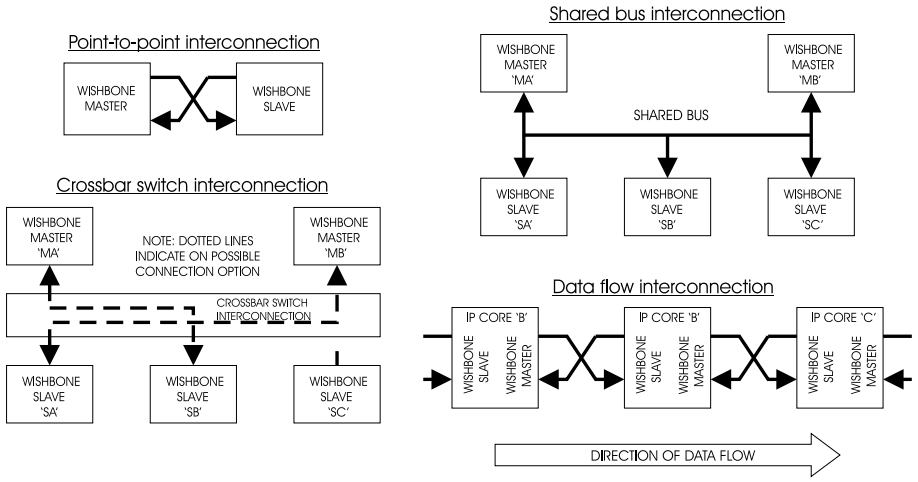

Figure 2.5: WISHBONE interconnect formats: *Point-to-point*, *crossbar switch*, *shared bus* and *data flow* [22].

2.4.2 Configurability

Both OCP and WISHBONE define a basic interface complying with the standards. This interface can be expanded with additional features suggested in the standards. OCP also supports what is referred to as *sideband signals* enabling test and control signals (interrupts etc.). The OCP test signals support both JTAG (Joint Test Action Group) [6], which is a standardized IEEE (Institute of Electrical and Electronics Engineers) scan test format, and a general purpose scan test interface.

Both OCP and WISHBONE interfaces are configurable as to data and address size.

2.4.3 Comparison

A preliminary study of the OCP showed that it was suitable for implementation in this project and was therefore chosen to save time. Further studies has shown that WISHBONE just as well could have been implemented, with the same result of portability. However, besides the mere functionality a few other factors have to be considered to compare the two standards.

Comparing the functionality they both excel at various areas. WISHBONE has a stronger interconnect format capable of the same and more than the OCP. While WISHBONE could be used as an extra layer above a different type of interconnect (as the OCP is supposed to), it is also possible to use only WISHBONE on the SoC. This could show to be a great feature for SoCs with simple communication demands.

OCP on the other hand has a much greater functionality than the WISH-

BONE including support for test and control signals.

For development of OCP components a design tool named CoreCreator [10] is available. The functionality of this tool has not been examined. WISHBONE is the favored protocol for the OpenCores [13] initiative. Therefore there are a lot of cores already available for the WISHBONE format including interconnect structures such as the crossbar.

Regarding both OCP and WISHBONE it is also important to consider the licensing. OCP is claimed to be an *open* standard, however to gain access to the design tool and to include OCP communication in a commercial design, it is necessary to become a member of the OCP society amounting to a minimum of $10,000 annually. WISHBONE is on the other *public domain*, i.e. absolutely free. This may change in the future but that is unknown at the moment. The cores published under the OpenCores initiative are covered by the *OpenIpCore* license [13] which comes in two formats: One resembling the *GNU GPL* (General Public License) and one resembling the *GNU LPGL* (Lesser General Public License) [4]. An important statement of the OpenIpCore license is, that hardware implementations of designs including such cores can be produced and sold under any license found suitable. However, source code for the entire design must be made public. This could show to be a quite inhibiting factor for many companies.

Both OCP and WISHBONE show great potential for SoC core implementation. The choice of which protocol to employ must be based on the demands of the design to implement correlated with considerations of the licensing. Another great factor is how the protocols are accepted by the SoC community. Already OCP is implemented in new designs by large corporations such as Nokia, Synopsys and Texas Instruments, indicating it could become an industry standard.

Chapter 3

OCP protocol

The OCP protocol has been chosen for implementation in the mini-core system, as described in section 2.4.3. This chapter elaborates on the short introduction already given, targeted at later implementation in the mini-core system.

All information about the OCP protocol stated in this chapter originates from [11].

3.1 OCP Architecture

The OCP protocol is a synchronous core-centric protocol (see section 2.1) defining a bus-independent and configurable interface between IP cores and on-chip communication interconnect.

OCP is based on MASTER/SLAVE communication as seen in figure 2.4. 3 types of signals are defined between MASTER and SLAVE:

- Data flow - used for the actual data transferring between cores.
- Sideband - control signals (reset, interrupt etc.).
- Test - enables scan test and JTAG [6].

The protocol requires an interface to contain a minimum of signals to be OCP compliant. These are all *data flow* signals and are seen in table 3.1.

3.2 Commands and Responses

A MASTER can issue one of 5 commands determining the transaction format between MASTER and SLAVE:

- Idle.
- Write.

Signal	Description
Clk	Clock.
MCmd	Command from MASTER to SLAVE.
MAddr	Address from MASTER to SLAVE.
SCmdAccept	Acknowledge signal from SLAVE to MASTER.
SResp	Response from SLAVE to MASTER.
SData	Data from SLAVE to MASTER.

Table 3.1: Signals necessary for minimum implementation of OCP interface.

- Read.
- ReadEx (read exclusively).
- Broadcast.

The *Write* and *Read* commands are self explanatory. *ReadEx* is an extension of the *Read* command. When a MASTER receives data as a response to a *ReadEx* command, the SLAVE has control over the addressed resource and the MASTER can therefore post a *Write* to change the specified resource. This is a convenient way to implement atomic *read-modify-write* transactions. *Broadcast* is used to tell a SLAVE to pass on received data to all remote targets connected "on the other side" of the SLAVE, i.e. the SLAVE core functions as a bridge to a different network or similar.

Read and *Write* are mandatory, while *Broadcast* and *ReadEx* are optional.

A SLAVE can post one of 3 responses to a command:

- No response - NULL.
- Data valid/accept - DVA.
- Response error - ERR.

3.3 Transfer Phases

A transfer between MASTER and SLAVE is initialized at a rising clock edge by the MASTER posting a command on the *MCmd* signal. If the command is either a *Read* or *ReadEx*, the SLAVE will post a response. If the command is either a *Write* or *Broadcast*, the SLAVE will only respond if *handshaking* is turned on, which is optional.

Within a transfer the various phases are ordered. For instance if the SLAVE is to respond, the MASTER must not post a new request until the SLAVE has responded. However it is possible to establish several transfer *threads* between MASTER and SLAVE as described in section 3.4.1.

3.3.1 Timing

Figure 3.1 shows a timing diagram for a valid OCP transfer. It is seen that the SLAVE raises the *SCmdAccept* signal (3rd clock cycle) before posting a response (6th clock cycle). The number of clock cycles from the request to the command accept is referred to as *request-accept-latency*. The number of clock cycles from the MASTER posting a request to the SLAVE responding is referred to as *request-to-response-latency*. For the transfer in figure 3.1, the request-accept-latency is 2 and the *request-to-response-latency* is 5.

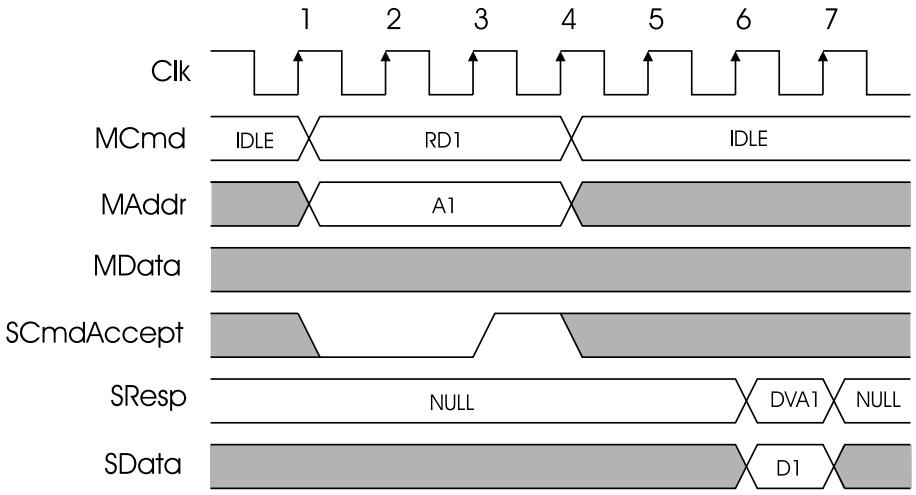

Figure 3.1: Timing diagram for an OCP *Read* transfer [11].
CC 1: MASTER posts *Read* command. CC 3: SLAVE acknowledges command. CC 6: SLAVE responds with data.

3.4 Extensions

The basic interface as described until now can be extended in a variety of ways. Some of the most important extensions are mentioned in this section. Figure 3.2 shows all the signals of the OCP protocol.

3.4.1 Threads

It is possible to establish several communication *threads* between MASTER and SLAVE. The transfer phases within each thread must be ordered as described in section 3.3. The phases in the different threads are however completely independent. A MASTER can post a request on one thread and while waiting for a response on this thread, post a request on a different thread.

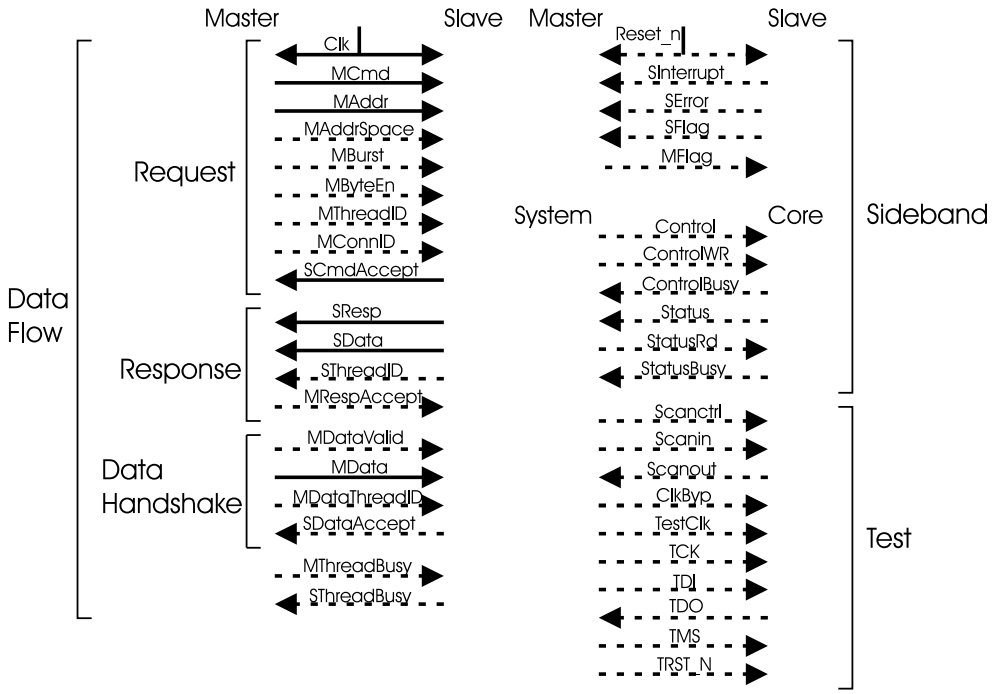

Figure 3.2: All OCP interface signals [11].

3.4.2 Burst

Bursts are used to link together a set of transfers, writing or reading a group of data. While in burst mode a signal is continually informing the SLAVE how much data remains to be read or written. There are 4 burst types available:

- Incrementing.
- Streaming.
- Custom burst that can be packed.
- Custom burst that cannot be packed.

The incrementing burst increments the address by the word size every clock cycle. This can for instance be employed for reading or writing blocks of a memory.

The streaming bursts leaves the address unchanged. Useful for sequential data transfers as for instance modems and Analog/Digital converters employ.

The last two burst types are configurable as to addressing etc. Packing refers to the fact, that data with a different word size than the chosen OCP data size can be transferred in a burst. If the word size is smaller than the

OCP data size more than one word could be transferred in a single burst packet - i.e. *packed*.

3.4.3 Test Signals

It is possible to add test signals to the interface. These are not connected in a MASTER/SLAVE format but will most often consist of scan chains through the entire interfaces. Two types of test signals are available, one set for custom scan design and one set conforming with the JTAG scan test standard [6].

3.4.4 Sideband Signals

Besides the mentioned extensions, it is also possible to add a few other features to the interface through the sideband signals.

These include a global reset, interrupts, flags and error passing between MASTER and SLAVE. Furthermore, it is possible for the environment (system computer or similar) to pass data directly to the MASTER/SLAVE of a core.

Chapter 4

Mini-Core System

A behavioral SystemC model of the mini-core system serves as test platform for the communication interfaces defined in this thesis. The implementation of this behavioral SystemC model is also part of the thesis. This chapter provides a description of the mini-core system as basis for the SystemC implementation.

First is described the purpose and architecture of the mini-core system. Following is a description of the FIR and IIR mini-cores, which currently are the only available mini-cores for the system. Finally is a description of the communication format employed in the mini-core system.

4.1 Overview

4.1.1 Background and Purpose

The mini-core system is a result of a joint effort at the Computer Science and Engineering department at institute of Informatics and Mathematical Modelling at Technical University of Denmark, including various master's and Ph.D. theses.

The purpose of the mini-core system was to create a programmable system dedicated to perform DSP, with the focus on low power consumption. The system is a heterogeneous multiprocessor system, meaning that it is a connection of a variety of different processor types, each processor type optimized for a specific task.

As each processor core is optimized for a specific DSP task, it becomes a compact element with a small instruction set, resulting in low power consumption. Hence the term mini-core.

The mini-core system was designed in VHDL at RTL (Register Transfer Level). The RTL design employs many low-power design methods resulting in a quite complex VHDL design. The VHDL design has been synthesized

into gate level and implemented on an IC. The IC contains 3 FIR processor mini-cores (section 4.2), 3 IIR processors (section 4.3) and an I/O (Input/Output) unit connected on a bus. This structure is also employed in the behavioral SystemC model. From now on this is the structure referred to by the term mini-core system.

As the mini-core design only exists as a quite complex VHDL design, a behavioral model of the system is implemented in SystemC facilitating easier design modification. The SystemC implementation is not directly based on the VHDL design due to the complexity of this design, instead it is based on the documentation of the mini-core system.

4.1.2 Architecture

The basic structure of the mini-core system is seen in figure 4.1. Data is passed from and to the mini-core system through the I/O unit. The processors are programmed through a configuration network connected to all the cores (not shown in figure). This enables downloading of binary programs, constants etc. before program execution. The mini-core system has a specific configuration mode in which this is carried out.

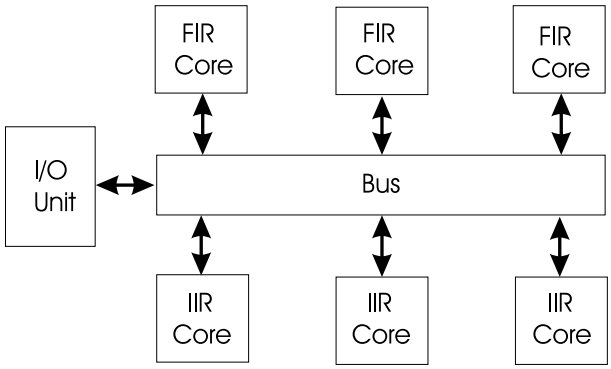

Figure 4.1: The mini-core system structure.

4.2 FIR Processor Mini-Core

The FIR mini-core is a programmable processor dedicated at implementation of FIR filters. The FIR mini-core is described in detail in [7]. Following is a shorter description of the FIR processor mini-core.

4.2.1 Instruction Format

The processor instruction format contains a 4 bit opcode, 4 bit flags and an *immediate* with customizable width. The destination/source register is also

contained in the flags section when necessary.

In the JMP instruction (table 4.1) the flags and the *immediate* are used together as the jump address.

The instruction formats are seen in figure 4.2.

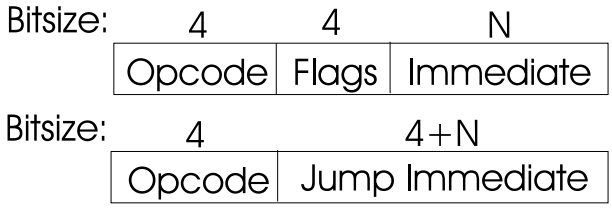

Figure 4.2: Instruction format of FIR processor. Top: Standard format. Bottom: Jump instruction format.

4.2.2 Instruction Set

The instruction set of the FIR processor is quite small with only 15 instructions. An overview is provided in table 4.1, for more information please refer to [7].

The specific action of some of the instructions depends on the state of the flags.

4.2.3 Flags

The processor contains 19 flags as seen in table 4.2. Only a subgroup of the flags are used by each instruction, the MACC instruction for instance only employs the COMP, CLR and FIN flags. It is therefore possible to dedicate only 4 bits to the flags.

4.2.4 Registers

The processor contains 8 registers, which are seen in table 4.3. All registers are available as source or destination for instructions.

4.2.5 Memory and Pointers

The processor contains 4 memories and 7 pointers into these memories. These are seen in tables 4.4 and 4.5.

4.2.6 Implementation of FIR Filters in the Processor

With instruction set, memory organization and pointers covered, the usage of the processor can be explained.

Instruction	Description
NOP	No operation.
BRA	Branch to instruction if register is equal to/different from zero.
JMP	Jump to instruction.
LOAD	Load from coefficient or delay-line memory into register.
STORE	Store from register in coefficient or delay-line memory.
MOVREG	Move value between register and delay-line memory. The difference between this and STORE/LOAD is the addressing format employed.
SWITCH	Switch filter. A number of filter structures can be stored in memory. This instruction chooses which is the actual filter.
LSET	Set low part of register.
HSET	Set high part of register.
ADDI	Add *immediate* to register. This can be done modulo the current filter length.
SUBI	Subtract *immediate* from register. This can be done modulo the current filter length.
MACC	Multiply accumulate. Multiply coefficient with value from delay-line and accumulate.
ASMACC	Add/subtract multiply. As MACC besides that a subtraction/addition is performed between **two** values of the delay-line. The result is used for the multiplication.
SEND	The value of a register is send to an external member of the network.
RECEIVE	A value is received from an external member of the network. The processor will stall until this value is received.

Table 4.1: Instruction set of FIR processor.

Flag	Description
CLR	Clear accumulator before MACC or ASMACC.
COMP	Also calculate complementary result (useful in some filters) in MACC or ASMACC.
FIN	Last MACC or ASMACC, demands special pointer updates
SUB	Subtraction in ASMACC.
ADD	Addition in ASMACC.
ZERO	In BRA, branch if register equal to zero.
NZERO	In BRA, branch if register not equal to zero.
MOD	Perform ADDI or SUBI with modulo.
CP	Use coefficient pointer register in instruction.
DP1	Use delay-line pointer 1 register in instruction.
DP2	Use delay-line pointer 2 register in instruction.
MACC	Use MACC register in instruction.
COMP	Use COMP register in instruction.
TMP1	Use TMP1 register in instruction.
TMP2	Use TMP2 register in instruction.
TMP3	Use TMP3 register in instruction.
DM	Use delay-line memory in STORE or LOAD.
CM	Use coefficient memory in STORE or LOAD.
REGDM	Transfer from register to delay-line memory in MOVREG.
DMREG	Transfer from delay-line memory to register in MOVREG.

Table 4.2: FIR processor flags.

Register	Description
CP	Register containing coefficient pointer.
DP1	Register containing delay-line pointer 1.
DP2	Register containing delay-line pointer 2.
MACC	Multiply accumulate register used by MACC and ASMACC.
COMP	Complementary register used by MACC and ASMACC.
TMP1	General purpose register 1.
TMP2	General purpose register 2.
TMP3	General purpose register 3.

Table 4.3: FIR processor registers.

Memory
Delay-line memory.
Coefficient memory.
Instruction memory.
Filter memory.

Table 4.4: FIR processor memories.

Register	Description
BCP	Base coefficient pointer.
CP	Coefficient pointer.
BDP	Base delay-line pointer.
DP1	Primary delay-line pointer.
DP2	Secondary delay-line pointer.
CURR	Pointer to current filter in filter memory.
PC	Program counter.

Table 4.5: FIR processor pointers.

The coefficient memory and the delay-line memory are together with the MACC and ASMACC operations used to create a FIR filter like behavior. The input values/samples are stored in the delay-line memory and the coefficients in the coefficient memory. The processor can contain a number of different filter structures (interchanged with SWITCH), and the parameters for these are stored in the filter memory. There are 5 parameters for each filter, which are shown in table 4.6:

Register	Description
LN	Filter Length.
BCP	Base coefficient pointer.
CP	Coefficient pointer.
BDP	Base delay-line pointer.
DP1	Primary delay-line pointer.
DP2	Secondary delay-line pointer.

Table 4.6: Filter parameters for FIR processor.

CP, DP1 and DP2 have been described as both registers and values contained in the filter memory, which can be a bit confusing. These values are in fact stored in the filter memory but are treated as registers by certain instructions.

The implementation of a simple FIR filter as seen in figure 4.3 will be done as:

```
;label
start:
;switch to actual filter
switch 0
; receive data on channel 0
receive tmp1, 0
;move sample into delay-line
movreg regdm, tmp1
```

```
;perform filter calculation for this sample
macc clr, 1   ; clear accumulation register
macc 2        ; next coefficient is 0, skip
macc fin, 1   ; last calculation
; no operation because of delay slot
nop
;send result
send macc, 1
;loop forever
jmp start
```

At execution time the filter memory (filter 0) has been loaded with the filter parameters. As has the coefficient memory to contain the coefficients used.

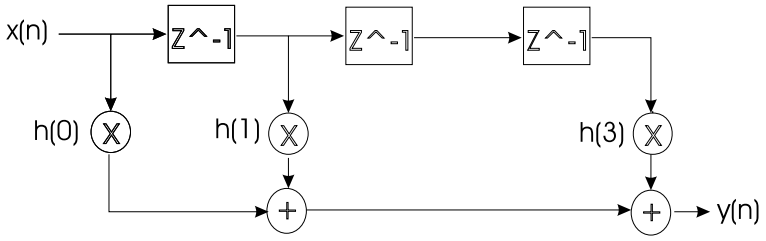

Figure 4.3: Simple FIR filter

4.2.7 Data Format and Precision

The data format of the registers depends on the instruction in which they are used. ADDI and SUBI conceives the registers as signed integers (including the MACC and COMP registers). The MACC and ASMACC instructions perform fixed point calculations, hence the value of the MACC and COMP register will contain a fixed point result after such an instruction.

The bit widths of registers and coefficients are configurable. The number of bits before the decimal point in fixed point numbers is also configurable.

The processor does not prevent or warn about overflow. In integer calculations the result will wrap around, and in fixed point calculations the result will saturate, meaning that values too large to be presented properly are presented as the largest possible value. However in many FIR filters coefficients are scaled to lie in the range of -1 to 1, in that way overflow does occur in the multiplications (but can occur in the accumulation).

4.2.8 Architecture of FIR Processor

Figure 4.4 shows the datapath of the processor, which is a 2 stage pipeline. Due to the method of implementation the MACC and ASMACC instruction have a delay slot of 1 unless followed by another MACC or ASMACC

instruction. All other instructions accessing the MACC or COMP registers also have a delay slot of 1.

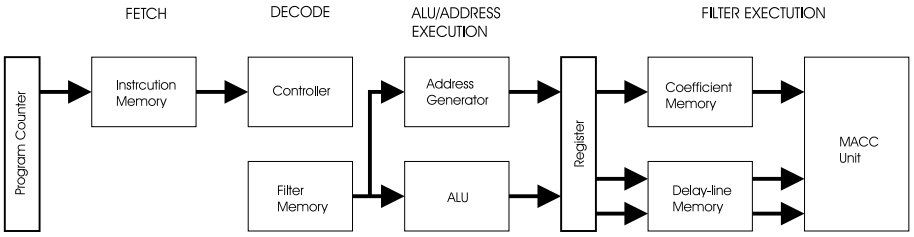

Figure 4.4: FIR processor datapath [15].

4.3 IIR Processor Mini-Core

The IIR mini-core is a programmable processor dedicated at implementation of Infinite Impulse Response filters. The IIR mini-core is described in detail in [15].

The IIR processor has 14 instructions but only a subset of the instructions will be described here. An examination of the instruction set showed that by utilizing a subset of these instructions, it is possible to create meaningful IIR filters. To simplify things it was chosen only to implement this subset of instructions.

4.3.1 Instruction Format

The IIR processor differentiates between 5 types of instructions formats. The subset of instructions includes type 1, 3, 4 and 5 instructions. The format of these can be seen in figure 4.5.

The instruction length is the same for all instruction formats. Therefore some of the bits are unused for certain format types. This is indicated by a blank box, which is found in all but format 5 type b. The fields to the right of such a blank box are right-aligned and the fields to the left left-aligned.

4.3.2 Instruction Subset

The implemented instruction subset of the IIR mini-core instruction set is seen in 4.7.

4.3.3 Biquad Instruction

The basis of the IIR processor is the BIQUAD instruction. A *biquad* refers to a second order FIR filter of direct form II, which can be used as a building

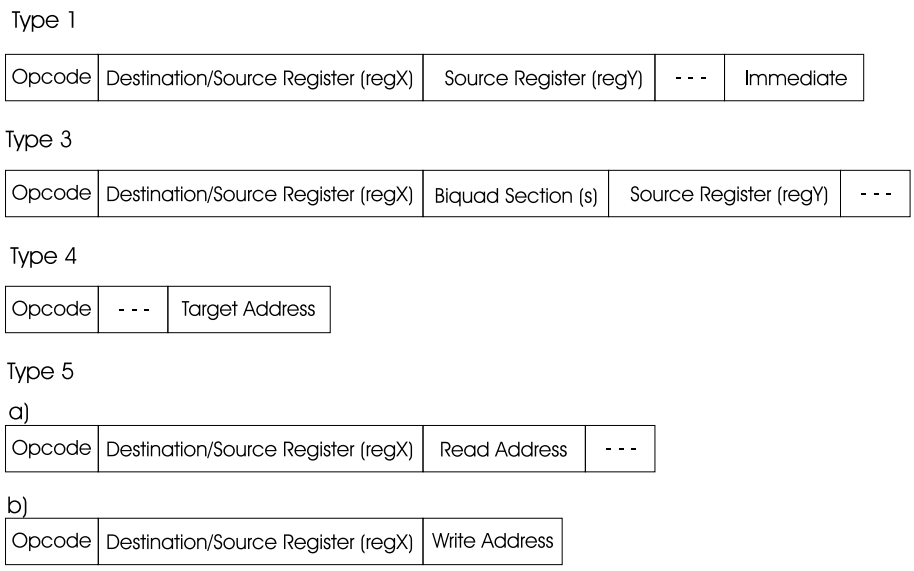

Figure 4.5: IIR processor instruction formats.

block for higher order IIR filters. The architecture of a biquad section is seen in figure 4.6.

As the biquad instruction is somewhat complex, it will be described in some detail here. The BIQUAD instruction computes the output of a biquad

Instruction	Description	Format Type
NOP	No operation.	1
ADD	Add $regY$ to $regX$ and store result in $regX$.	1
SUB	Subtract $regY$ from $regX$ and store result in $regX$.	1
MOV	Move contents of $regY$ into $regX$.	1
SHFRR	Shift $regY$ right the amount of times indicated in *immediate* and store in $regX$.	1
SHFLR	Shift $regY$ left the amount of times indicated in *immediate* and store in $regX$.	1
BIQ	Compute a biquad section and store result in $regX$.	3
JUMP	Jump to instruction.	4
RECEIVE	A value is received from an external member of the network.	5a
SEND	The value of a $regX$ is send to an external member of the network.	5b

Table 4.7: Instruction set of IIR processor.

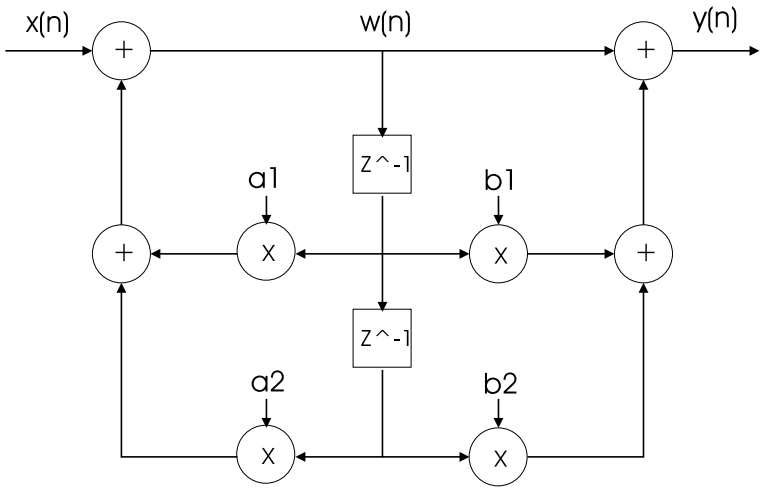

Figure 4.6: Biquad section.

section, updating all pointers, registers etc.

The output of a biquad section can be expressed as (with reference to figure 4.6):

$$w(n) = x(n) + a_1 * w(n-1) + a_2 * w(n-2)$$
$$y(n) = w(n) + b_1 * w(n-1) + b_2 * w(n-2)$$

This can be directly implemented in the behavioral model. The IIR processor can incorporate more than one biquad section. With s being the section index, the pseudo code for the BIQUAD instruction is:

$$w(s)(n) = regY + a_1 * w(s)(n-1) + a_2 * w(s)(n-2)$$
$$regX = w(s)(n) + b_1 * w(s)(n-1) + b_2 * w(s)(n-2)$$

This is a direct implementation of the mathematical expression, with $x(n)$ stored in $regY$ and $y(n)$ stored in $regX$.

4.3.4 Datapath

The datapath of the IIR mini-core (which is a 3 stage pipeline) is seen in figure 4.7.

Similar to the FIR mini-core, the IIR mini-core contains an instruction memory, a coefficient memory and a register file. It furthermore contains the *biquad register file* used for the BIQUAD instruction, a *shift-add* unit and the *dual multiply-accumulate* unit - used in the BIQUAD instruction.

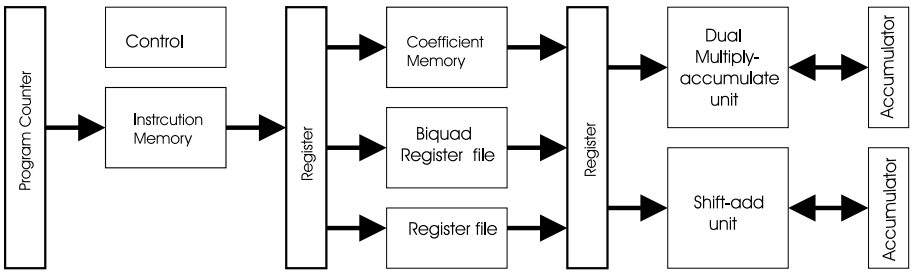

Figure 4.7: IIR mini-core datapath [15].

4.3.5 Miscellaneous

The data format is the same as for the FIR mini-core. Fixed point for BIQUAD and integer for ADD and SUB.

The SEND and RECEIVE instructions are similar to them in the FIR mini-core instruction set.

4.4 Communication

This section describes the communications format of the mini-core system, originating from [9].

4.4.1 Core SEND and RECEIVE Instructions

The mini-cores operate internally with a SEND and a RECEIVE instruction, which are used to transfer data between cores.

The mini-cores expect the SEND to be carried out and do therefore not wait for an acknowledge. The RECEIVE on the other hand waits for the data to arrive. So unless data already has arrived for the core (but has not been received by it), the core will stall to wait for data.

The SEND instruction indicates the destination address and the register from which to take the data. The RECEIVE instruction indicates the source address and the register to place data in.

The destination and source addresses of these instructions indicate a *communication channel.*

4.4.2 Communication Channels

The mini-cores each have a given number (denoted N) of input channels available, which enable them to differentiate between the sources of the input data. These channels are referred to a *local channels.* The cores can read from 1 of the N local channels, which one is indicated in the RECEIVE instruction.

The channel indicated in the SEND instruction must somehow be mapped to the proper mini-core and local channel. The simplest method is to statically link the cores with their channels. Such that when sending to for instance channel 17, it is globally known that this is core X, local channel Y. This is exactly what is done in the mini-core system. With N channels per core, core i is linked to global channels $N(i-1)$ to $Ni-1$. This structure is seen in figure 4.8.

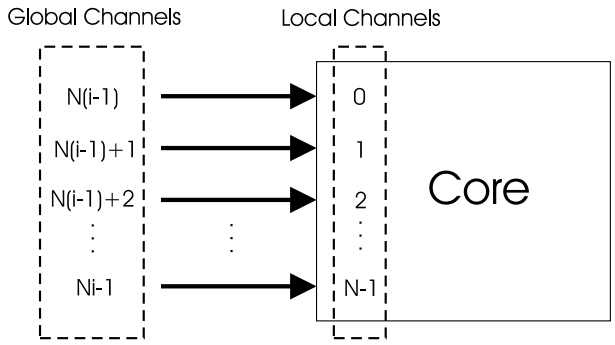

Figure 4.8: Global to local channel mapping in cores.

A more flexible method is to download local channel tables for each core. This way a core sending to 2 other cores does not need to know the addresses of every channel in the network.

The mini-core system has 4 local channels per mini-core. Table 4.8 shows the channel addresses for the mini-core system.

Core	**Channels**
FIR 1	0-3
FIR 2	4-7
FIR 3	8-11
IIR 1	12-15
IIR 2	16-19
IIR 3	20-23
OUTPUT	24-27
INPUT*	28-31

Table 4.8: Communication channels of the mini-core system. * The INPUT core does not receive data.

4.4.3 Network Structure

As previously mentioned the mini-cores are connected on a bus. The detailed structure of the mini-core system is seen in figure 4.9. Basically a mini-core

requests the interface either to send or receive data. This request is (if necessary) passed on to the bus node, which is connected with the other bus nodes of the system.

The protocols between the various units are explained separately.

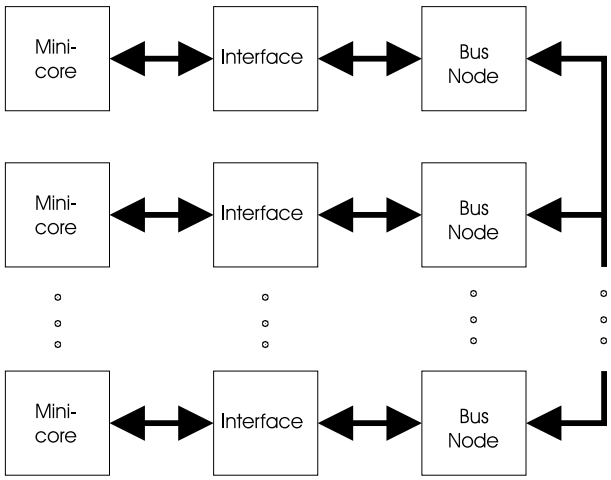

Figure 4.9: Network structure of the mini-core system.

4.4.4 Mini-Core to Interface Communication

The connections between a mini-core and an interface are seen in figure 4.10.

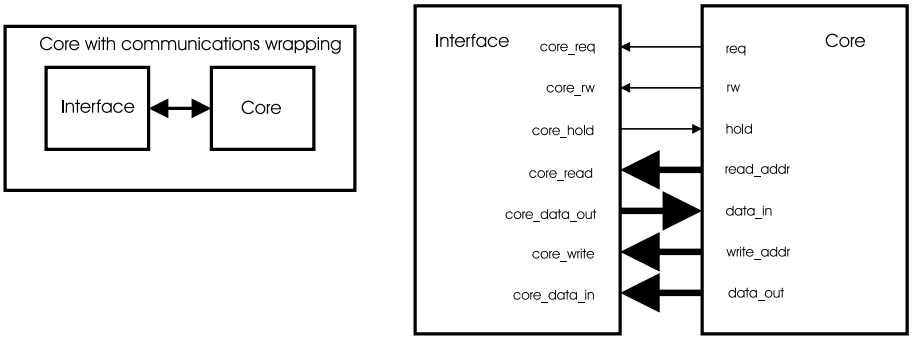

Figure 4.10: Left: Basic core, interface and communication wrapper structure. Right: Connections between interface and core.

A core requests to either send or receive data (indicated by *rw*) by raising the *req* signal. At the same time the read/write address is specified (the source/destination channel) with *read_address*/*write_address*.

The interface contains a buffer for both sending and receiving, for receiving there is a buffer per local channel. The purpose of this buffering is to

smoothen the data transfers, thus avoiding *peaks*. The size of these buffers is optional and is off course a trade off between network performance and overhead in interface size. In the mini-core system these buffers have the size of one data word.

If a core requests to receive data from an empty buffer or attempts to send data with the send buffer full, the interface must stall the core until the buffer status has changed. This is achieved with the *hold* signal. The mini-core SEND and RECEIVE instructions wait to see, if they caused a stall. As both the mini-cores and the interfaces are trigged on the positive clock edge, a hold is received in the 3rd clock cycle, therefore execution of the SEND and RECEIVE instructions each take 3 clock cycles.

When sending, *data_out* contains the value to the other core. When receiving data is passed to the *data_in* signal of the core.

A mini-core together with an interface can be regarded as a core with a communications wrapper (see section 2.2), as shown in figure 4.10.

4.4.5 Interface to Bus Node Communication

The connections between interface and bus node are seen in figure 4.11.

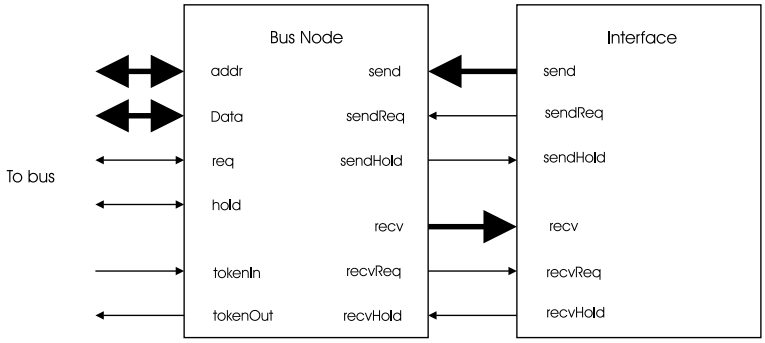

Figure 4.11: Bus node with connections to bus and interface.

If the bus node receives data for the core, it raises the *recvReq* signal. The *recv* signal contains both data and channel address. If the buffer of the interface is full, it raises the *hold* signal, until the core receives data. It is necessary also to stall the mini-core trying to send data, which is achieved by stalling the entire network.

If the interface requests to send data to another mini-core, it raises the *sendReq* signal, and *send* contains both the data and the destination channel. Because this may cause a network stall it must wait, until it is sure the data has arrived before processing new requests. This takes approximately 5 clock cycles.

4.4.6 Bus Structure

The nodes are connected to the *addr* (address) and the *data* bus. The *req* and *hold* signals shown as bi-directional signals are in fact separate signals, with the *holdIn* (*reqIn*) for all the nodes being the logic OR of all the *holdOut* (*regOut*) signals.

The *tokenIn* and *tokenOut* signals are utilized for handling the bus write permission as explained in section 4.4.8.

4.4.7 Bus Protocol

The protocol of the bus is quite simple. The node with bus permission has control over the bus for one clock cycle. Data and address is written on the negative clock edge, together with the request.

All nodes scan the address bus to see if there is valid data on the bus (the *reqIn* signal is high), and if the data is destined for them. If so the data is stored and passed on to the interface. No acknowledge is send from destination to source.

All nodes can set the *hold* signal stalling the entire bus, as explained in section 4.4.5. All nodes are idle during a hold.

If the network is stalled until a mini-core receives data, a deadlock occurs if the mini-core stalling the network wants to send data before receiving. To avoid this the node stalling the bus must not itself be stalled. This has to be handled locally in the node to avoid excessive bus logic.

4.4.8 Bus Arbitration

Because all the nodes are connected to the same bus, it is necessary to determine which node has the permission to the bus. In this particular design it is handled locally in the nodes. The main reason for this is to make scaling easier.

The arbitration is carried out by passing a token through the nodes. The token passing is started every clock cycle, with one of the tokens being the initiator. If a node receives this token and has data for the bus, it will not pass the token to the next node, and so have bus permission this clock cycle - next clock cycle it is token initiator. This is sometimes referred to as *round-robin* scheduling, since the nodes are asked one at a time in a circular fashion.

To cause a minimal time delay the token is not passed on the bus but on a combinatorial signal ring containing all nodes, as seen in figure 4.12. This is a simple scheduling format, which does not put extra load on the bus. A serious disadvantage is however, that it makes scaling more troublesome as the routing of this combinatorial ring can cause problems. If however (as currently in this design) the amount of nodes is relatively small, this should not pose any problems.

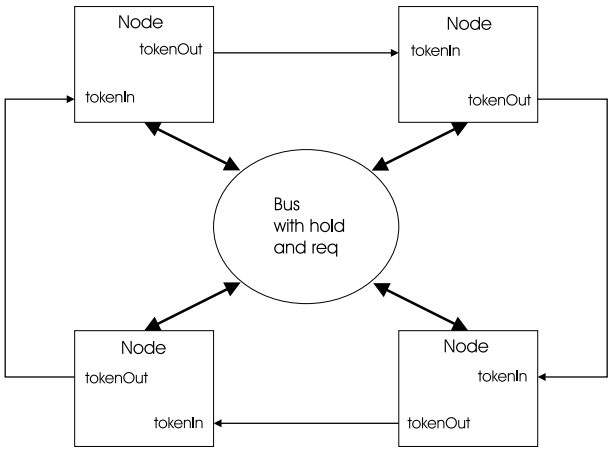

Figure 4.12: Token ring for bus nodes.

4.4.9 Hold Signals

To recapitulate, the situations resulting in holds are:

- A core requests a RECEIVE and the specified receive buffer of the interface is empty. The core is stalled until data arrives.
- A core requests a SEND and the send buffer of the interface is full. The core is stalled until the interface is allowed to send data.
- An interface receives a SEND request from another core, and the receive buffer for this channel is full. The **entire** network (all nodes) is stalled until the core reads data from this channel.

Chapter 5

Implementation of Mini-Cores

This chapter explains the SystemC implementation of the FIR and IIR processor mini-cores. As a foundation general processor architecture is shortly described, as is the use of HDLs.

Furthermore, two additional mini-cores developed to simulate data transferring with surroundings of the mini-core system are described.

5.1 Architecture of Dedicated and General Purpose Processors

In [16] a processor is described as the combination of control unit and data path, which in a standard personal computer leaves memory and I/O devices. More specifically a processor must be able to execute instructions contained in a program and in such a way perform calculations and interact with memory and I/O units of the system. Instructions and data consumed/produced are stored in the memory of the system. In the von Neumann model employed in most personal computer processors, a single memory is used for both instructions and data.

In a processor one instruction is executed every clock cycle, with the PC (Program Counter) determining which. The instructions are normally read sequentially but some instructions are able to modify the PC, hence altering the order of execution. A processor normally consists of these major components:

- Instruction and data memory.
- Dedicated registers such as program counter and pointers into memory.
- General purpose registers to store often used values.

- Arithmetic Logical Unit (ALU) which can perform a variety of arithmetic and logic operations (such as additions, ANDs etc.).
- Control unit decoding instructions and setting control signals for other units.

Very simply described the actions performed sequentially in a single clock cycle are: Update program counter, read instruction, decode instruction, read from registers and/or memory (if necessary), perform ALU calculation and write to registers and/or memory (if necessary).

5.1.1 General Purpose vs. Dedicated Processors

Processors are most often separated into either general purpose or dedicated processors. As the names implies, this indicates whether the processor is used in a system performing a specific operation or various operations.

The CPU (Central Processing Unit) in a personal computer is a general purpose processor, since the computer can execute all sorts of programs, e.g. operating systems, games, word processors etc. The processor on the graphics card in a personal computer is on the other hand most often a dedicated processor. Since the assignments for this processor are well defined, the instruction set can be tailor made for calculations etc. regarding graphics. Dedicated processors are found in many places, e.g.: Mobile phones, routers, washing machines and game consoles (Playstation etc.). The processors in this project are to be described as dedicated processors.

5.1.2 Programming of Processors

Every processor has a well defined instruction set, which can be utilized in creating programs for it. The instructions are stored in the memory as a bit pattern (machine code) but this is quite tedious to create maually. Above machine code in the hierarchy is assembler language. This basically consists of a mnemonic for each instruction (e.g. *add* for addition), which is easier for a programmer to remember than the binary code for the same. A compiler translates the assembler code into binary code. In recent years assembler programming has been replaced by high level languages like C, C++, Java etc. Code written in such a language is translated into machine code via assembler code.

5.1.3 Pipelining of Processors

As mentioned, one instruction is executed each clock cycle. This begins on a rising clock edge by changing the PC, hence reading a new instruction. The next instruction cannot be read until all calculations have been performed, and the results have been stored in registers or memory. The time this

takes is called the delay and denoted T. The delay differs from instruction to instruction but T is specified for the slowest instruction. The throughput of the processor indicates how many instructions are performed per second, this is found as f=1/T (Hz).

It is sometimes desirable to increase the throughput of a processor, which increases the speed of the processor if the instructions are unaltered. Since most processors perform various operations in a sequential order when executing an instruction, the different components (ALU, register etc.) lie unused some of the clock cycle. If this can be resolved such that the components can be divided into blocks only used in a portion of the clock cycle, these blocks can be separated by registers. The delays of these blocks are separately smaller smaller than the original delay. The instructions traverse through the blocks, with different instructions in each block in a given clock cycle. This increases the throughput of the processor and is referred to as pipelining. If the data path is divided into 3 blocks it will be referred to as a 3 stage pipeline.

The result of transforming a single-cycle processor into a 5 stage pipeline is shown in fig 5.1.

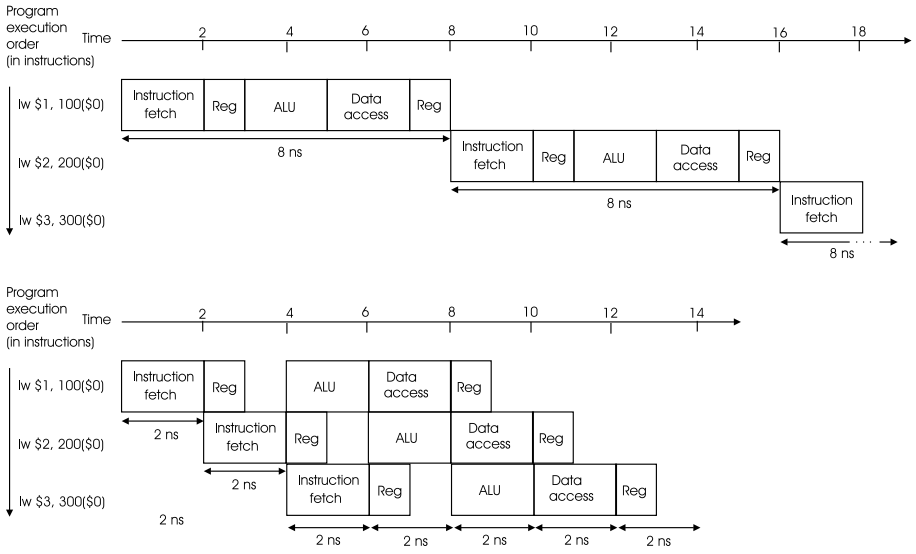

Figure 5.1: Pipelining of processor. Top: Single-cycle version. Bottom: 5 stage pipeline version [16].

The disadvantage of pipelining is the increased complexity, since an instruction can depend on data written by a previous instruction, which may still be in the pipeline (and so have not calculated or stored the result yet). The simplest way to solve this problem is to introduce *delay slots*, which states that after a certain type of instruction, N clock cycles have to pass (delay

slot of N) before a certain type of instruction can occur. This can be incorporated into the assembler compiler, which will examine for dependencies between the instructions and insert *no operation* instructions if necessary.

5.2 Designing Processors Using HDLs

HDLs can be used for designing processors. The result can be either an ASIC implementation, a processor capable of running in a FPGA (Field Programmable Gate Array) or simply a model used to simulate the operation of a processor. The last can be useful for several reason, for instance during the process of developing a processor, where simpler models can be used to test principles or perhaps the interaction with other systems.

Before commencing the design itself some considerations must be made as to the choice of which HDL to use and at what abstraction level to work.

5.2.1 Abstraction Level

In figure 5.2 is shown an overview of the various levels of system modeling. When using HDL one usually is situated in the functional modeling either at RTL or algorithm level. Models constructed at algorithm level are usually referred to as *behavioral*.

As the name implies RTL indicates that the entire design is constructed by registers and combinatorial blocks. A simple counter could for instance consist of a N bit register and a combinatorial N bit adder. The adder itself could then be a behavioral or RTL model.

Whether to employ a RTL or behavioral description of a design very much depends on what the product of the design is meant to be. With reference to figure 5.2 an ASIC is situated on the structural transistor level, whereas a FPGA design is situated at the structural gate level. The translation from the chosen design level of abstraction into the product's level of abstraction has to be performed by a design tool of some origin. The translation from RTL to gate and transistor level is a very well proved process. Translation from behavioral level or RTL into gate level is often referred to as *synthesis*.

Years of research have resulted in design tools, that are very well aware of how to recognize and implement a register, half adder etc. from RTL code. The process of translating an algorithm into RTL is on the other hand not as well proved. A lot of research has been carried out in recent years but problems arise because of the multitude of RTL descriptions capable of implementing a given algorithm, making it difficult for a design tool to recognize the optimal RTL implementation of an algorithm.

In recent years design tools have been created facilitating synthesis of behavioral designs adhering to certain guidelines. However, the results vary a lot, and at least some basic knowledge about RTL modeling is necessary in

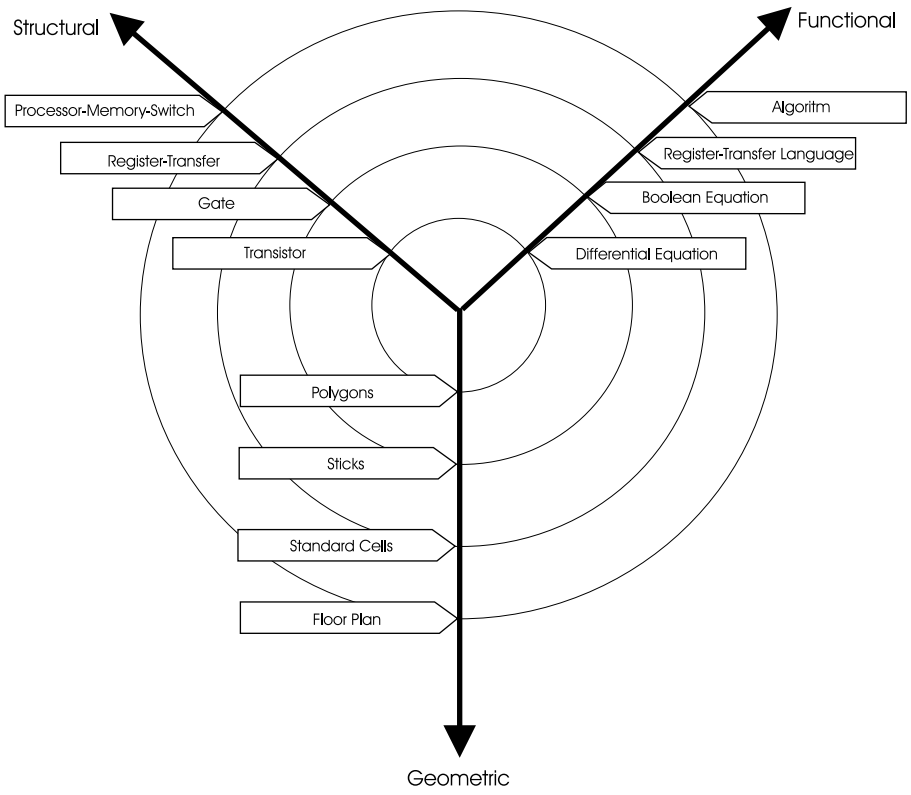

Figure 5.2: Abstraction levels [2].

order to comprehend and fix errors in the synthesis result due to the design tool misunderstanding the behavioral description.

5.2.2 Choice of HDL

In this project SystemC is the chosen HDL. This is in an effort to exploit the possibilities and fallacies of the language and to gain experience with the language.

This choice is of pure educational reasons. Disregarding this aspect, a more objective comparison of the possibilities within the area of HDLs must be performed prior to choosing the HDL of a design. A brief overview of the HDL market is provided in the following.

There exist a wide range of HDLs but the three currently most used are: VHDL, Verilog and SystemC.

VHDL is by far the most employed HDL in Denmark and the rest of Europe, whereas Verilog is the most employed HDL in USA. The difference between these two languages lies mostly in the syntax, and some synthesis tools even translate VHDL into Verilog before synthesizing. The choice

between VHDL and Verilog will therefore most often be the result of regional and corporate practice and policy.

SystemC is a rather new language (first release in September 1999) based on C++. At RTL level the difference between System and VHDL/Verilog is not that significant. In some cases however, it is evident that the SystemC syntax has not been designed specifically for hardware design (but adopted from C++), making some things quite complex compared to VHDL/Verilog. When moving into a higher level of abstraction SystemC has some definite advantages. SystemC is in fact a collection of packages for C++, and therefore it also incorporates all other facets and packages of C++. This is especially useful when creating behavioral models (e.g. for testing), since these can be created at a very high level of abstraction.

One of the main reasons for developing SystemC was that many large designs start with a functional description in C/C++ (or similar high level programming languages), and from there are translated into a HDL. By using C++/SystemC for the entire design this process should be greatly simplified.

The major downside of SystemC is it's immaturity. It has not existed long enough to be equipped with the same supply of design tools as VHDL and Verilog. Furthermore (and perhaps worse) is that the documentation of SystemC is somewhat incomplete. A lot of questions have to be answered by traversing through Internet forums and the like. Books have started to surface on the subject but it is still very far from as well documented as VHDL and Verilog.

5.3 HDL Implementation of FIR and IIR Mini-cores

The implementation methods of the FIR and IIR mini-cores are similar and therefore explained as one in this section.

5.3.1 Choice of Abstraction Level

The SystemC implementations of the mini-cores are used as models of the original mini-cores, which means that the external behavior must be identical. However as a starting point they need not to be synthesizable, therefore the simplest form of implementation is a behavioral description.

As previously described (section 4.2.8) some instructions have a delay slot of 1. Modelling this sort of behavior will make the behavioral model significantly more complex. It has therefore been chosen to disregard this behavior in the models. Some of this behavior can be simulated by insertion of NOPs in the program of execution at relevant points.

5.3.2 Algorithm of Implementation

When viewed upon as a behavioral system, the FIR and IIR mini-cores have the following distinct processes within a clock cycle.

- Update program counter.
- Fetch instruction.
- Decode instruction.
- Read from memory/registers (depending on instruction).
- Perform ALU operation (depending on instruction).
- Update pointers (depending on instruction).
- Write to memory/registers (depending on instruction).

The registers are implemented as simple variables. Whereas the memories are implemented as arrays. Reading/writing from/to registers and memories (including pointers) is therefore very simple and can be combined with the ALU operation in the algorithm.

A simplified pseudo code description of the algorithm is:

```
// declarations of memory and register variables/arrays

if (rising clock edge) {

  PC++;
  load_instruction();
  demux_instruction();
  switch (opcode) {

  case EXAMPLE:
    load_register_or_memory();
    ALU_calculation();
    write_result_in_reg_or_mem();
    if (BRA || JMP)
      update_PC();
  end case;
  ...
  }
}
```

5.3.3 Data Format

The data format employed by the various registers and memories complies with the original design. As a standard is used signed or unsigned integers, which in SystemC can be bit manipulated in the same manner as bit vectors. This spares a great deal of type conversion when performing ALU operations.

For fixed point calculations SystemC's incorporated fixed point type is used. This is instantiated with bit width (*wl*), number of bits before decimal point (integer word length, *iwl*), method of quantization (*o_mode*) and how to handle overflow (*o_mode*). The syntax is:

```
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> x;
```

The parameter *n_bits* is used only for certain methods of overflow handling and not in the one in question.

The quantization and overflow method of the original designs has been adopted, meaning rounding to nearest presentable number and saturating at overflow. The *wl* and *iwl* are configurable as described in next section.

To indicate the chosen quantization and overflow methods in the instantiation predefined SystemC constants are used. The constants for the methods of choice are *SC_RND* and *SC_SAT*. To use these constants it is necessary to include the following line in the code:

```
#define SC_INCLUDE_FX
```

This must also be in any file including such a file before the inclusion itself.

The format of the fixed point numbers is 2's complement. The bit expression of a fixed point can be converted to an integer/bit vector (and vice versa) by bitwise copying. This is used to convert between integers and fixed point in the design.

5.3.4 Programming of Mini-Cores

In the original design the processor is connected to a configuration network to enable programming when situated in an ASIC or FPGA. Since the SystemC design is not to be synthesized, the configuration network is replaced by simply loading the memory arrays from files before starting program execution.

There exists an assembler compiler for the FIR and IIR mini-cores, which creates the binary code specifically suited for programming via the configuration network. This format does not comply with the FIR processor instruction format, therefore two programs to change the binary code have been created, one for the FIR and one for the IIR mini-core. The programs are named *postasm_fir* and *postasm_iir*.

The assembler compiler also accepts the various constants (coefficients etc.) within the assembler file. The behavioral SystemC models demands the instructions, coefficients etc. to be in separate files. To ease this a program (named *coeff*) has been created to translate the coefficients from hexadecimal format (as employed in the assembler) to bit strings (as employed in the behavioral mini-core system).

When compiling one of the original assembler programs for the behavioral SystemC model, on has to:

- Move coefficient declarations from assembler file into separate coefficient file.
- Translate coefficients into correct format with *coeff* program.
- Compile assembler program with original compiler.
- Pass binary output through either *postasm_fir* or *postasm_iir*.
- For FIR mini-core, create the contents of the filter memory containing filter lengths etc. manually.

5.3.5 Configurability of Mini-Cores

The registers and memories are configurable as to size and bit width. In the original VHDL design this has been incorporated using *generics*, simply described as configurable constants. The counterpart to this in SystemC is *templates* originating from C++. By using templates it is possible to pass parameters to a class when instantiating it. This is actually what is done when telling the SC_FIXED what bit width etc. to use.

The design does however contain a great deal of configurable parameters, therefore it has been chosen to gather some of these in a single file (*declarations.h*) to simplify the instantiations.

The parameters included in the templates are the filenames used to load the memories as they vary when instantiating a multitude of processors and the bit width of the communication signals. Since the template structure already exists, it is a simple task to add the other configuration parameters if desired.

The task of passing string (such as filenames) as parameters to a template is somewhat tricky in C++/SystemC. The string cannot be passed directly as:

```
my_fir<"filename">;
```

The string must be contained in a named variable declared outside the scope of the function in which the instantiation is performed, like this:

```
const char *nameoffile = "filename.bin";

my_function() {

          my_fir<nameoffile>;
}
```

When employing templates one can encounter a great deal of problems trying to divide a class into two files (.cpp and .h) as is normally done when programming in C/C++. Therefore all classes utilizing templates in this design are contained in a single .cpp file.

5.4 Mini-Cores Simulating Environment I/O

Two mini-cores have been constructed to simulate the I/O with the environment of the mini-core system. These are named *input* and *output* and simulate the environment sending and receiving data. Both mini-cores have an exterior similar to the FIR and IIR mini-cores.

5.4.1 Input Mini-Core

This mini-core simulates the environment sending data to the mini-core system. The data to be send is read from a file. The mini-core is told from which file to read data and how often to transmit it.

It is as the other mini-cores configurable regarding the bit width of the communications signals

5.4.2 Output Mini-Core

This mini-core simulates the environment receiving data from the mini-core system. It constantly waits for data to receive. Received data is instantly written to a file, which file is configurable. It interprets the received data as fixed point and writes it to the file as decimal values.

It is as the other mini-cores configurable regarding the bit width of the communications signals

Chapter 6

Implementation of Mini-Core System Communication

This chapter describes the various communication formats implemented in the SystemC behavioral mini-core system, beginning with the format employed in the original design. Furthermore is described the implementation of the OCP wrappers followed by the definition and implementation of a general purpose interface for the interconnect. Finally is described how the communication modules have been modified in such a way they are able to log various network traffic information.

6.1 Original Communication Format

This section describes the implementation of the original mini-core system communication format [9] described in section 4.4, with reference to figure 4.9 showing the entire structure. The structure and signals described have also been employed in the SystemC implementation.

6.1.1 Interface and Core Wrapping

The interface and core wrapping structure is seen in figure 4.10. This structure is unchanged in the implementation. The interface is named *simple_interface*, and the wrapped cores are named *fir_wrap*, *iir_wrap*, *output_wrap* and *input_wrap*

The interface contains 1 SEND buffer and 4 RECEIVE buffers (one for each local channel), with a buffer length of one data word.

Inputs are sampled at rising clock edges and outputting occurs at falling clock edges. This way it is not necessary to implement semaphores in order to avoid reading from and writing to a buffer simultaneously.

Data placed in the SEND buffer is automatically transmitted to the node on the next falling clock edge, unless the node is stalling the interface.

Data and address sizes are configurable through template arguments.

6.1.2 Bus Node

The bus node is seen in figure 4.11. The SystemC implementation has been named *bus_node*.

The node employs two buffers, one for data to the network and one for data received from the network, both with the size of one data word. Inputs are sampled at rising clock edges, and outputting occurs at falling clock edges for the same reason as with the interface.

The token chain (as seen in figure 4.12) is implemented by a combinatorial function and two synchronous functions. One of the synchronous functions is only active if the node is the token initiator in the current clock cycle. It passes the token (a logic '1') to the next node in the ring at the rising clock edge. The combinatorial function is triggered by the *tokenIn* port and an internal signal indicating if there is data to transmit. The combinatorial function constructs a combinatorial chain passing the token through the nodes. The second synchronous function resets the token chain at falling clock edges, removing all tokens.

Data and address sizes are configurable through template arguments. Furthermore is the address and if the node is the initial token initiator provided in the template.

6.2 OCP Implementation

This section describes the considerations of the OCP protocol implementation in the mini-core system together with the implementation itself, based on chapter 3.

6.2.1 Considerations on Implementation of OCP

It was chosen to retain as much of the original design structure as possible thus saving time. This was achieved by altering the interface and node to communicate by the OCP protocol as seen in figure 6.1. This results in a structure capable of interchanging OCP wrapped cores, and facilitates employing the wrapped mini-cores in other systems supporting the OCP protocol.

The demands of the mini-core communication are quite simple. In fact only SENDs are passed through the interfaces onto the bus, as RECEIVEs are latched internally in the interfaces. This actually shows that the basic

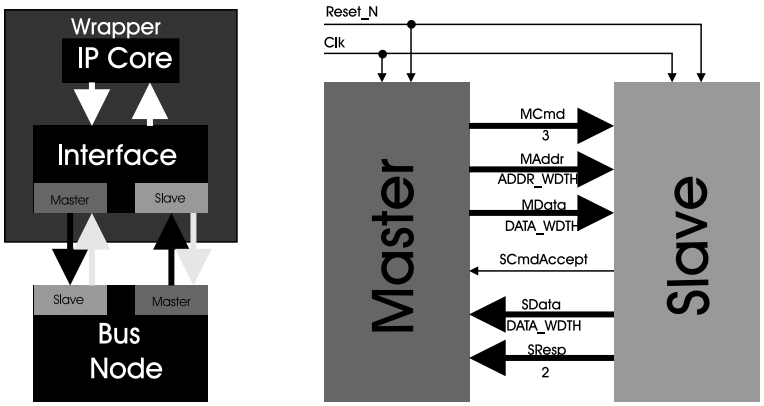

Figure 6.1: Mini-core with OCP wrapper connected to bus node.

OCP interface (added a global reset) more than satisfies the demands of the original mini-core system.

6.2.2 Timing

The timing specification of the OCP is described in section 3.3.1. It was chosen to make the *request-accept-latency* and the *request-to-response-latency* identical, simply stating that the acknowledge and the response from the SLAVE are posted in the same clock cycle. The latencies are a minimum of one clock cycle, there is no upper boundary. This timing format is fully compliant with the OCP format. Timing diagrams for *Read* and *Write* transactions following this specification are seen in figure 6.2.

6.2.3 SystemC Implementation

It was chosen to disregard the *Read* command in the SystemC implementation, as it is superfluous. The reason is that the cores do not support being requested to transmit data, and data send by a core is transmitted as quickly as possible to the destination. Support for the *Read* command is however required by the OCP protocol and can be implemented with a small effort as all signals are available. If implemented the *Read* request should be posted to the OCP SLAVE of the node but not passed further on to the interconnect, as it will pose unnecessary strain on the network. Reaching the SLAVE it shall wait until data destined for the local channel is received.

If implementing a more complex communication methodology in the mini-core system (for instance if constructing other mini-cores demanding this), the *Read* command could be used to synchronize data transferring between cores.

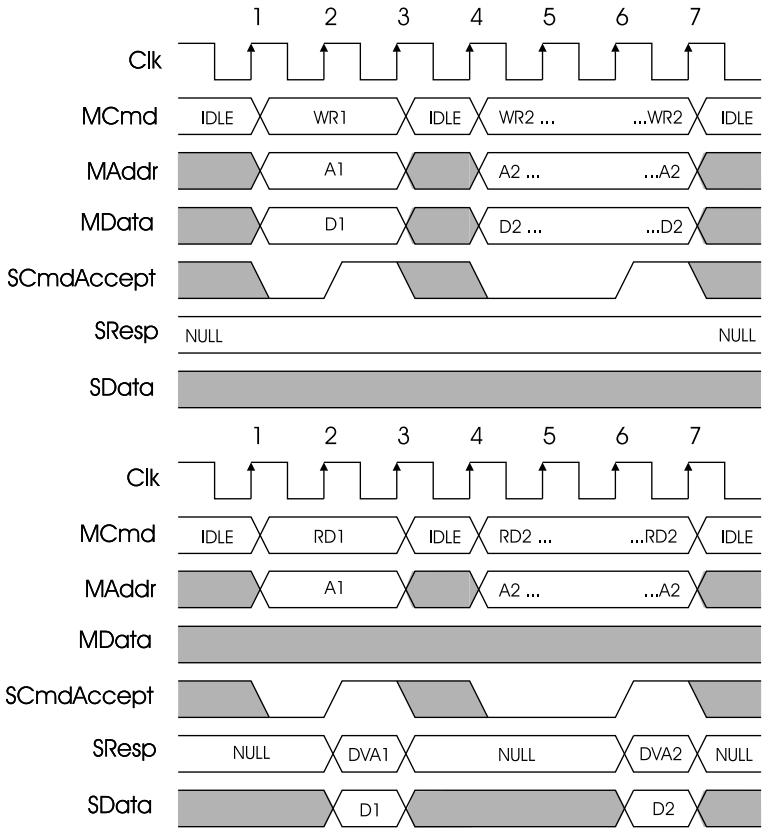

Figure 6.2: Timing diagrams for OCP implementation. Top: *Write* transaction. Bottom: *Read* transaction. Diagram format adopted from [11].

The node, interface and wrapper have been altered to implement the OCP protocol, they have been renamed *ocp_node*, *ocp_interface* and *<core_type>-_ocp_wrapper*. The node and interface have each been supplied with a OCP SLAVE and MASTER but have otherwise been left unchanged.

The OCP MASTER and SLAVE have been implemented according to an API (Application Programming Interface) described in [5]. The API is a result of a collaboration between Nokia, Texas Instruments, Synopsys and Sonics. With the support of such large corporations the API could develop into a standard, which is the reason for employing it in this project. The SystemC implementation of the API has not been released as of yet, even though it was due to release in the end of 2002. Therefore the functions described by the API have been implemented as found reasonable. This way it should be possible to exchange the API functions of this project with the proper ones when released.

The API declares functions supporting the *Read* and *Write* commands. As the *Read* command is unsupported, these functions have not been implemented. Furthermore the API functions support separate acknowledge and response cycle, which does not occur in design. Therefore the functions specifically targeted at this behavior have not been implemented.

The API differentiates between 4 communication abstraction layers, with RTL being the lowest layer (layer 0). The *transfer layer* (layer 1) has been chosen for this implementation as it is behavioral, cycle-true and employs the same signals as the RTL implementation. This conforms with the previously designed communication modules.

6.3 General Purpose Interconnect Interface

This section describes the definition and implementation of a general purpose interconnect interface in the mini-core system, allowing to change the interconnect format without affecting the rest of design. This enables benchmarking of algorithms on various interconnect formats (bus, ring etc.) with a small effort.

6.3.1 Method

To be able to freely change the interconnect format, all specific information about the interconnect format must be placed in the interconnect module itself. The interconnect module must be equipped with a general purpose interface, similar to the cores. By introducing such a general purpose interface, the interconnect can be conceived of as wrapped, resulting in a structure similar to that seen in figure 2.3.

In the mini-core system this implies modifying the node, as it is the only module that is interconnect specific.

6.3.2 Interconnect Interface

The interface of the interconnect module must enable connected modules to request data transfers through the interconnect. To do so the modules must supply data, address and a request. The interconnect must be able to transfer data to a module, therefore also supplying data, address and a request. Furthermore the interconnect must be able to stall modules attempting to transfer data but not allowed. Likewise the modules must be able to stall the interconnect in the same situation.

A simple general purpose interface fulfilling these demands is seen in figure 6.3. The interface is based on the general purpose core interface.

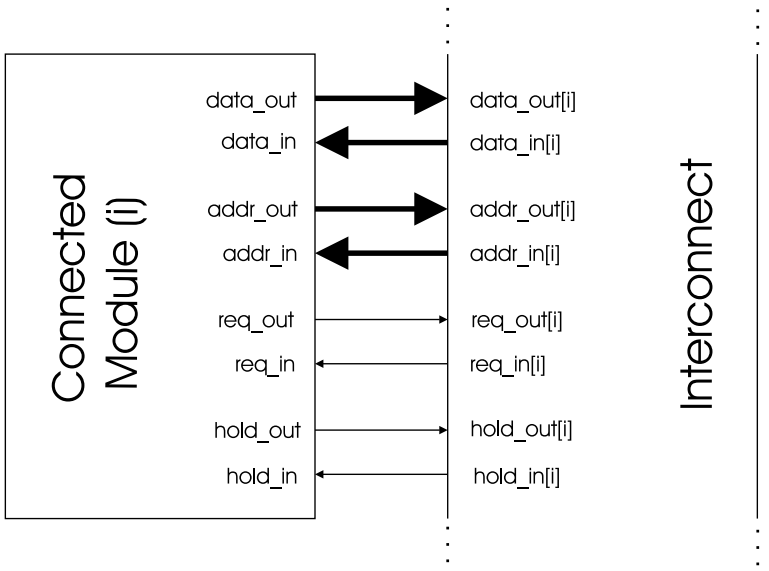

Figure 6.3: General purpose interface between interconnect and connected modules.

6.3.3 Resulting Nodes

The modifications of the existing *ocp_bus_node* are quite simple. The resulting module is named *ocp_generic_node*.

Most importantly all the bus arbiting logic is removed, as arbiting no longer is performed locally in the nodes. Otherwise the changes mainly consist of modifying signal names, as the functionality is almost identical.

6.3.4 Bus Interconnect Module

To illustrate the principles mentioned a bus interconnect module has been implemented, named *bus_generic*. An arbitrary number of modules can be connected to it (>1) allowing data transfers between them employing a shared bus behavior.

The signals of all the modules are transferred to the bus interconnect as arrays of signals (i.e. array of data signals, address signals etc.). The interconnect includes arbiting, which decides what module is the next in line to transfer data. All modules are granted permission in a round-robin manner as in the original mini-core system. All modules requesting bus permission but not granted it are stalled for the current clock cycle.

As the ports of the interconnect are implemented as arrays of signals, the reading/writing of data from/to the relevant module is easily achieved by the use of pointers into the signal arrays.

The module is configurable as to the number of modules connected, data

and address width and the amount of communication channels per module.

6.3.5 Other Interconnect Modules

Employing the same array structure as the bus interconnect module other structures can be implemented. A ring structure could be implemented by mapping the *data_out* of module *i* to *data_in* of module *i+1* and likewise with the rest of the signals. Some additional logic is required to enable data passing through the modules.

6.3.6 Timing

The important aspect of the employment of a behavioral interconnect module for benchmarking is that the exterior behavior is unchanged, such that it is a cycle-true simulation of the "proper" interconnect.

By employing the method described here, the behavior of the original design is altered by the addition of a clock cycle to the transfer latency, because the interconnect module is itself synchronous. This behavior has been disregarded as it is deemed insignificant. To rectify this behavior an asynchronous interconnect structure could be implemented, this will however be somewhat more complicated.

6.4 Logging of Traffic Information

To avoid extracting traffic information from complex and confusing simulation wave forms, functions for logging of essential traffic information have been constructed.

For SoCs the most important traffic information is normally situated around the cores and around the interconnect.

6.4.1 Core Traffic Information

Regardless of interconnect format etc., an important aspect is how often the various cores request transfers, and how often they wait for these transfers to be accomplished. This information can be logged in the core wrappers, where the various request and hold signals are available.

The *<core_type>_ocp_wrappers* of the mini-core system have been modified, such that they can log the amount of clock cycles the core is requesting to receive data, requesting to send data and being stalled. This information is written to a file. There is no differentiating between whether stalled cores are waiting to send or to receive data. To implement this it will be necessary to store the last request type.

Whether to output the traffic information and to which file is stated in the template.

6.4.2 Interconnect Traffic Information

The interconnect also contains valuable traffic information. The *bus_generic* has been modified as to output this information.

For the bus structure the useful information is how often a module drives the bus, how often it is stalled and how often it is stalling the modules connected. This information is written to a file. Furthermore, a run-time report logging the bus transaction every clock cycle (if any) stating clock cycle, bus driver and hold driver is created. Whether to output this information is supplied through the template.

When logging traffic information from other interconnect structures the implemented functions cannot be directly employed. The structure itself is reusable but for each interconnect format, it is important to decide which information is relevant.

6.4.3 File Formats and Post Processing

The runtime report is a text file, with each line containing information on a clock cycle with a bus transaction.

The bus interconnect module and each of the wrappers output the traffic information to separate files in unsigned integer format. The information from these files can be gathered to create useful diagrams showing the traffic distribution of the system. This can be automatized for instance using Matlab. Two such diagrams created in Matlab are seen in figures 7.4 and 7.5.

Chapter 7

Test of Mini-Core System Implementation

This chapter describes the test of the SystemC behavioral implementation of the mini-core system including the various communication alterations and expansions.

7.1 Purpose

The testing executed in this project is not a complete test of all implemented modules. The reason is that the mini-core system has already been thoroughly tested. A complete test in this project could discover minor bugs in the SystemC implementation but this has been deemed insignificant as long as the overall functionality is correct.

Therefore the test is designed to show, that a behavioral system capable of performing the same tasks as the original mini-core system has been constructed. Furthermore, the test will show that the various modifications of the communication format have left this functionality unchanged.

7.2 Method

The simplest method to test the mini-core system is to download test programs in the FIR and IIR mini-cores, letting the INPUT and OUTPUT mini-cores supply and store test data.

During the development of the mini-core system and communication modifications, various test programs have been constructed testing small portions of the design. Instead of including all these small tests, it was chosen to test the design by executing an authentic algorithm specifically constructed for the mini-core system.

7.3 Test Algorithm

The algorithm chosen to test the system is part of a digital equalizer, which amplifies or attenuates the various frequency bands of a signal. The test algorithm named *filterbank* divides the incoming signal into 7 frequency bands, utilizing FIR filtering. The frequency bands are outputted on separate channels. The algorithm is originally described in [14] and modified in [7] to employ two FIR mini-cores instead of only one, it is the second format that is used for the test.

The 7 frequency bands are added in the IIR mini-cores before written to a file. It is thus possible directly to compare the input data of the system with the output data. The input data consists of a 10 period sinus signal created in Matlab. As the input data consists of a single sinus wave, all data should ideally be transferred through one of the channels. Since all channels are added, the test does not ensure that the sinus is situated in the correct channel. However since the algorithm has already been tested [7], [14], it should be ensured that data is divided and added properly.

To execute this algorithm the behavioral mini-core system must me able to execute programs, perform FIR filtering (which includes the complex MACC and ASMACC of the FIR mini-core) and to send data between the various mini-cores.

The structure of the algorithm on the mini-core system is seen in figure 7.1.

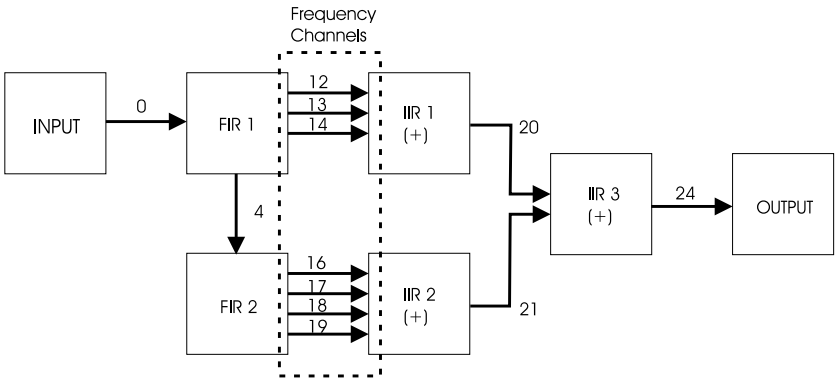

Figure 7.1: Layout of *filterbank* algorithm. The numbers by the connections indicate the channel number. Channels 12-14 and 16-19 contain the 7 frequency band. FIR 3 is unused.

7.4 Results

As mentioned the input and output data of the algorithm are directly comparable. Plots of input and output data in both time and frequency domain

are seen in figure 7.2 and 7.3. It is quite obvious that after a stabilizing period (for the FIR filters), the output closely resembles a sinus with same amplitude and period length as the input signal. It has not been possible to compare this result with the original mini-core system, as no test data has been available. However this result clearly indicates proper implementation of the behavioral mini-core system, however not renouncing minor bugs.

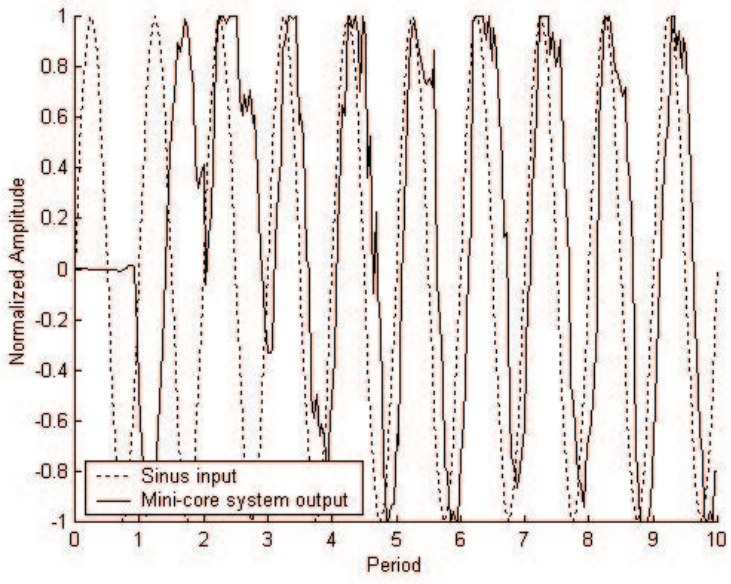

Figure 7.2: Comparison of sinus input and *filterbank* output in time domain.

The test has also been carried out for each of the communication modifications with the same result.

Figures 7.4 and 7.5 show the traffic information of the algorithm, which is unchanged for the various communication modifications.

Figure 7.4 shows that none of the cores stall the network. The reason is that no core transmits data faster than the receiver accepts it, thus not filling the buffer of the receiver. This situation can be provoked by informing the INPUT core to transmit faster, causing FIR 1 to stall the bus. However the system will in its current implementation most probable not return from this state. The reason is that all cores - including FIR 1 - are stalled, and with FIR 1 most likely wanting to transmit data before receiving data, this will cause the system to deadlock. Figure 7.4 also shows that FIR 1, FIR 2 and IIR 3 in about 20% of the clock cycles requesting to send data are stalled. This is not because the bus i overburdened but simply because they coincidentally request the bus in the same clock period.

In figure 7.5 it is seen, that all cores except INPUT spend much time

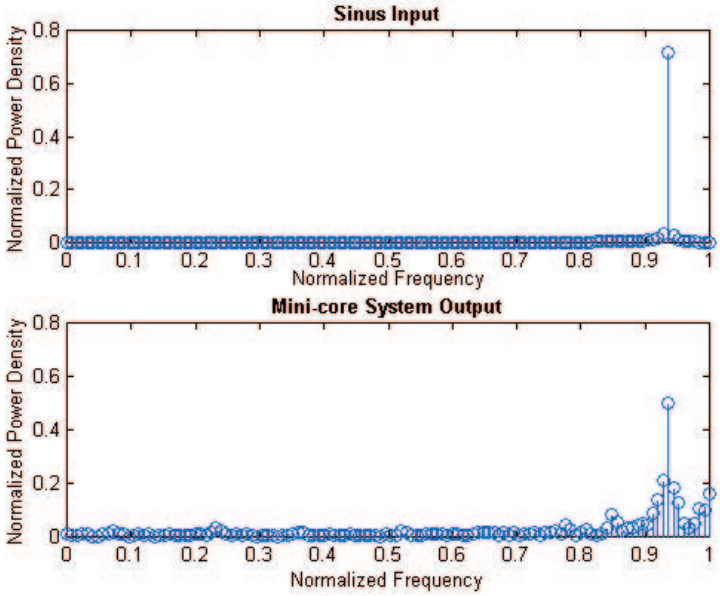

Figure 7.3: Comparison of sinus input and *filterbank* output in frequency domain.

waiting for data. With no send holds, all holds indicate cores waiting to receive data. This is partly because FIR 1 waits for data from the INPUT core causing the rest of the cores also to wait. Furthermore IIR1, IIR2, IIR3 and OUTPUT perform very little data processing compared to FIR1 and FIR2 causing extra wait cycles.

A small section of the run-time report is shown below:

```
Run-time report for simulation of mini-core system.
Shows:
-Clock cycle
-Node driving bus
-Hold, node with full buffer stalling bus.
For each is shown the address of the node responsible for
the signal. -1 indicates that no node is setting the signal.

Clock cycle      Node driving bus        Hold from node
55               7                       -1
69               0                       -1
88               0                       -1
95               0                       -1
99               1                       -1
```

The run-time report looks slightly different when employing the *bus_generic* structure due to the additional transfer latency of one clock cycle.

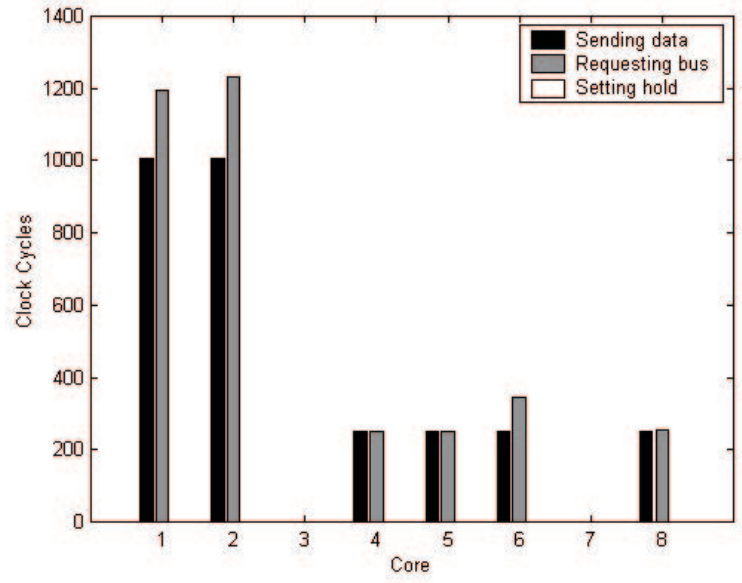

Figure 7.4: Traffic information for bus interconnect. Core 1-3: FIR 1-3. Core 4-6: IIR 1-3. Core 7: OUTPUT. Core 8: INPUT.

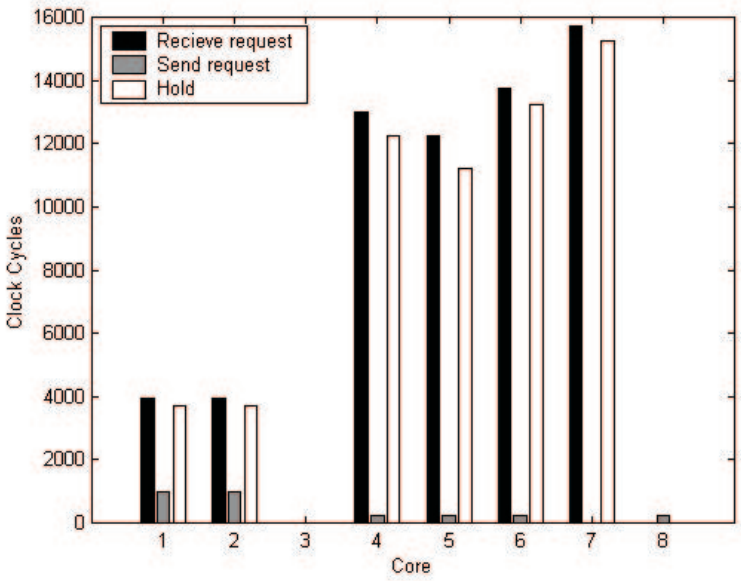

Figure 7.5: Traffic information for mini-cores. Core 1-3: FIR 1-3. Core 4-6: IIR 1-3. Core 7: OUTPUT. Core 8: INPUT.

Chapter 8

Discussion

This chapter summarizes and discusses the results and experiences achieved in this project, both concerning the mini-core system but also more general observations.

First is a summary of the results achieved. This is followed by a description and discussion of the mini-core system as implemented in this project. Finally is a discussion of the applicability of the implemented communication principles in general.

8.1 Results

4 major accomplishments have been achieved in this thesis:

- Construction of a behavioral SystemC model of the mini-core system.
- Implementation of the core-centric OCP protocol in the mini-core system, based on research into the field of SoC on-chip interconnect protocols.
- Definition and implementation of a general purpose interconnect interface together with a bus interconnect compliant for this structure.
- Construction of functions capable of extracting traffic information from the system.

The resulting outlay of the mini-core system is seen in figure 8.1. Both cores and interconnect are equipped with general purpose interfaces and supplied with OCP communication wrappers. This allows both interconnect and cores to be changed easily. A behavioral bus interconnect module has been constructed verifying the interconnect/OCP wrapper structure.

Furthermore, both core wrappers and the behavioral bus interconnect have been equipped with the ability to log information about the data traffic of the system.

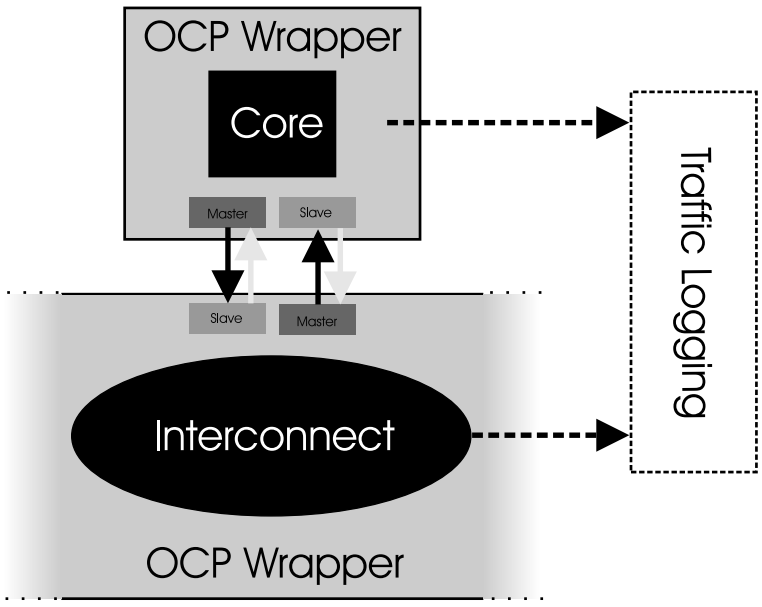

Figure 8.1: Mini-core system with OCP wrapping of cores and network.

The functionality of the system has been tested by the use of an algorithm developed for the mini-core system. The test shows that the wrapping employed leaves the functionality of the system unchanged, except for increasing the communication latency of the system by one clock cycle. The latency increase is a result of implementing the behavioral bus interconnect as a clock synchronized module and not of the wrapping itself.

8.2 Behavioral Mini-Core System

This section discusses the possibilities of the behavioral mini-core system as constructed in this thesis. As it shows the same external behavior and is capable of executing the same programs as the original system, it is the obvious test platform for further development of the mini-core system.

Core Development Test Platform

If new cores are to be developed for the mini-core system, their functionality can easily be tested by creating behavioral models of them. This of course causes an extra workload, however the behavioral models should pose a valuable basis for a later synthesizable implementation. Furthermore, a behavioral model generally is able to replace a substantial part of the documentation, for instance rendering a pseudo code description superfluous.

Algorithm Test Platform

The behavioral mini-core system can also be employed as test platform for algorithm development. It is safe to say that the simulation time for the behavioral SystemC model is significantly lower than for the RTL VHDL model, although it has not been possible to compare the simulation time of the two models. Furthermore the logging of traffic information eases detection of bottlenecks, deadlocks etc.

Communication Test Platform

The wrapper structure employed in the behavioral mini-core system eases changing both the core-centric communication protocol (to for instance WISHBONE) and the interconnect format. This facilitates implementing other interconnect structures (ring, cube etc.) without affecting the remaining system. It is also possible to introduce extra communications layers such as AMBA as interconnect. These features make it possible to benchmark performance on a variety of communication structures before choosing a communication architecture for an actual implementation.

8.2.1 Further Work

An elaboration on some aspects of the behavioral mini-core system implementation could be desirable:

- A rigorous test of all the implemented modules.
- Development and execution of algorithms more communication intensive than the existing mini-core system algorithms.
- Construction of a library of behavioral interconnect modules.
- A study of the possibility that more complex transmit formats (burst, threads etc.) can improve system performance.
- An elaboration on the traffic information logging, for instance performing statistically calculations.
- Experiments with other protocols such as WISHBONE.

8.2.2 Synthesis and Hardware Implementation

The mini-core system already exists as a synthesizable VHDL design. So unless porting the entire system from VHDL to SystemC, there is no reason to create a synthesizable SystemC system. If however porting the mini-core system to SystemC (or the behavioral system to VHDL), the result would be a flexible development platform capable of graduate refinement of modules.

Synthesizable modules could be tested together with behavioral modules, thus decreasing simulation time.

The interfaces and wrappers developed in this thesis are under all circumstances desired to be synthesizable, regardless whether the modules are to be constructed in SystemC or VHDL. As synthesis of the design constructed in this thesis was not a direct objective, no effort has been made to implement synthesizable SystemC code. However the OCP MASTER and SLAVE employed in the interface and node (i.e. the wrappers) contain simple synchronous state machines, and do actually not differ significantly from a synthesizable implementation.

The behavioral interconnect structure will however pose problems in a hardware implementation. The array structure utilized in the behavioral bus implemented is not synthesizable. The objective of the format employed in the behavioral interconnect module is however not hardware implementation. It is designed to easily interchange behavioral interconnect models, thus deciding on the optimal interconnect structure based on algorithm benchmarks. A synthesizable interconnect can be introduced by supplying it with wrapping complying with the interface between node and interconnect. This interconnect could be a industry standard like the AMBA bus or, it could be a specifically designed interconnect.

The data logging to file is of course not synthesizable but this feature can be removed through the template.

8.3 General Observations

Even though the work in this thesis has been based upon the mini-core system, the resulting architecture is suited for a variety of different SoC designs. The steps of defining a core interface, an interconnect interface and implementing a core-centric protocol (e.g. OCP) can be employed in most SoC systems with the same result.

The major advantages and disadvantages of employing the communication wrapper structure and traffic information processing described in this thesis are listed below.

8.3.1 Advantages

Design Concurrency

By dividing the SoC design into cores and interconnect structure independent of each other, the system can to a great extent be designed concurrently. This makes a decrease in design time possible.

Decreased Simulation Time

Simulation of synthesizable modules (often RTL modules) and back annotated modules is often quite time consuming, due to the complexity of the modules. By constructing behavioral models of cores and interconnect structure, it is possible to test these complex modules together with behavioral modules, decreasing simulation time.

The cores do not need to be cycle-accurate internally to perform functional tests, as the wrappers translate requests into cycle-accurate transactions.

Increased Portability

Implementing a wrapper structure for both cores and interconnect makes it easier both to import cores not specifically designed for the system (IP cores) and to reuse constructed cores in later designs.

Decreased Workload

By employing an industrial core-centric communication format such as OCP or WISHBONE, the design workload can be decreased for two reasons. First because various initiatives are taken to ease the design of SoCs employing such protocols. For OCP there exists both a design tool CoreCreator [10] and an API [5] is under construction. WISHBONE is chosen as the communication protocol of the OpenCores [13] initiative, where various core designs are publicly available. The second reason for decreased workload is a decrease in design documentation. By conforming to a common communication format, little documentation of the SoC communication needs to be constructed.

Simplified Benchmarking and Debugging

Logging traffic information at cores and interconnect significantly simplifies both benchmarking and debugging of a SoC design compared to simulation wave forms.

Decreased Interconnect Dependency

By wrapping the interconnect the remaining system does not depend on the choice of interconnect format. It is therefore possible to decide on this format concurrently with designing the remaining system. By utilizing a library of behavioral interconnects, the decision of interconnect format can be based on benchmarks of the SoC for a selection of the formats. Again the traffic logging helps this benchmarking.

8.3.2 Disadvantages

Overhead

By introducing communication wrapping as described in this thesis, there is a risk of introducing a design overhead. This can result in both increased die area, decreased system speed and/or decreased power efficiency. The overhead of introducing for instance OCP in a system is impossible to predict, as it will depend on the system in which it is introduced. However, the configurability of the OCP interface can help to minimize the overhead introduced into a system.

The mini-core system employs a minimal OCP implementation. Determining the overhead requires synthesizing and comparing the original mini-core system and the OCP version, which has not been done. However a simple comparison of the OCP structure with the original design is possible.

The SLAVE/MASTER structure requires more signals than the original core to node connection. This is mainly because no data is passed from SLAVE to MASTER but the signals for this are mandatory. The bit width of the connections between wrapped core and node is for the original mini-core system (with reference to figure 4.11):

$$2(DATA\_WIDTH + ADDRESS\_WIDTH + 2)$$

and for the OCP format (with reference to figure 6.1):

$$2(2 \cdot DATA\_WIDTH + ADDRESS\_WIDTH + 6)$$

With a data width and an address width of 16 bits, this yields a bit overhead of approximately 60% for the wrapped core to node connection. If the original mini-core system is to be expanded to support cores being requested to transmit data, this can be achieved by adding two R/W signals indicating if the core requests/is requested to SEND or RECEIVE. There is however no need to add extra data signals, as the existing data signals can be used. If a core is requested to SEND by another core while attempting to SEND at own initiative, arbiting in the core wrapper must decide the priority. This method will only necessitate 2 extra bits for the core to node interface, resulting in a bit width of:

$$2(DATA\_WIDTH + ADDRESS\_WIDTH + 3)$$

Compared to the OCP structure this is still an overhead of approximately 54% with 16 bit data and address.

The logic overhead of implementing OCP will mainly arise from support

of the *Read* command, as it will result in a larger state machine. However the state machine will still be fairly simple, and the logic overhead should be minimal compared to the entire design.

Licensing

By implementing standardized protocols licensing restrictions are encountered, as briefly described in section 2.4.3. There is a financial cost of utilizing the OCP protocol for industrial design, whereas the WISHBONE standard is free of charge but requires designs employing WISHBONE to be open source. Likewise other SoC communication formats are protected by a variety of licenses.

8.4 Design Flow

This section provides basic guidelines for implementing the wrapper structure employed in this thesis in other designs. These guidelines are based on the experiences of the mini-core system implementation.

The 3 major steps of implementation are:

- Wrap cores with core-centric protocol of choice.
- Wrap interconnect with same core-centric protocol.
- Implement data logging functionality.

The choice of core-centric protocol depends on a variety of issues such as functionality, overhead and licensing. These aspects have been discussed earlier and will not be treated further.

8.4.1 Core Wrapping

To wrap the cores with a core-centric protocol, the cores must be equipped with a general purpose interface. The format of this interface will of course depend on the communication demands of the cores. In this thesis the core interface of the original mini-core system was reused as it was found suitable. The interface only enables cores to perform simple SENDs and RECEIVEs, and the cores can only be requested to RECEIVE data. The interface is generally suitable for other systems with the same communication demands. If the cores are to perform more complex communication, such as being requested to SEND or being able to burst etc., this will demand additional signals added to the interface.

The main purpose of the core wrappers are to convert between the format of the general purpose core interface and the core-centric protocol. As

these two protocols normally not are synchronized, the core wrapper must be supplied with buffers. The buffers serve to help avoid cores being stalled because of data peaks. The size of the buffers must be determined as a trade off between logic overhead and core stalling. In the mini-core system the buffers only have a size of one data word but generally it is desirable to make the buffer size configurable, such that it can be changed according to the expected data rate of the system.

8.4.2 Interconnect Wrapping

Wrapping of the on-chip interconnect is slightly more complicated than wrapping of cores. In this thesis a behavioral interconnect module with a general purpose interface quite similar to the core interface has been declared. A module (node) performs the conversion between core-centric protocol and the general purpose interface of the interconnect. However this method is only directly applicable to interconnects equipped with this general purpose interface.

If a industrial interconnect format is to be employed, it is necessary to convert between the core-centric and the interconnect communication format. Performing this conversion in a single module will result in the smallest overhead (**Method A**). If however the conversion goes from core-centric to general purpose format and from general purpose format to interconnect format (**Method B**), this will result in an increased overhead but can result in a decreased workload. The reason for a possible decreased workload is that it might be desirable to construct a library of interconnect wrappers for a variety of interconnect formats and core-centric protocols. If this library is to contain wrappers converting between M core-centric protocols and N interconnect formats, **Method A** necessitates the implementation of $M \cdot N$ modules, while **Method B** only necessitates implementation of $M + N$ modules.

8.4.3 Data Traffic Logging

The main purpose of data traffic logging is to ease extracting valid information about the communication profile from a simulation. As traffic logging results in a large amount of data to be stored, it is generally not useful in hardware implementations, as it will demand large amounts of memory or excessive data transferring to a control computer. Therefore modules capable of logging data traffic, should be configurable as whether to perform it.

In general the most interesting information about the data traffic of a SoC is: How often do the cores perform transactions, how often do the cores wait and how burdened is the interconnect.

The obvious place to log core transactions is in the core wrappers. The

mini-core system logs when the cores request to SEND or RECEIVE data, when the cores are requested to RECEIVE data and how often the cores are stalled. This could be expanded by logging why the cores are stalled, i.e. whether they are waiting to SEND or RECEIVE data. This will of course have to be adapted, if the core interface becomes more complex, logging information concerning bursts, threads etc. The approach will however remain the same.

There are 2 suitable places to log interconnect traffic information: In the interconnect wrapper and in the interconnect itself. The simplest method is to log the traffic information in the interconnect itself (as done in the behavioral bus module), because all information is directly accessible. This is of course not possible unless the interconnect is designed locally. If the interconnect is an external design, the traffic logging can be placed in the interconnect wrapper.

What information to log very much depends on the interconnect format, and it can become quite extensive for some of the more complex protocols. However the main objective must be to register how often the interconnect is used and how often it stalls cores. This provides information enabling detection of bottlenecks and deadlocks.

It is recommendable to limit the post processing of the traffic data in the system itself, instead storing it as "raw" data as to decrease simulation time. The data can be post processed by programs suitable for performing arithmetic operations on large amounts of data, an obvious choice being Matlab (as used in this thesis).

The traffic data of the mini-core system was simply read from files and printed in diagrams. This showed quite satisfactory because the communication of the executed algorithm is deterministic. However, if the system contained (pseudo) random elements, it could be desirable to perform statistical processing of the logged data.

Chapter 9

Conclusion

A behavioral model of the mini-core system has been implemented in SystemC. SystemC was chosen as HDL of purely educational reasons, and a short comparison with VHDL/Verilog is provided in section 5.2.2. The implementation was based on a study of the various components of the mini-core system, which include the FIR mini-core, IIR mini-core and the communication modules, all described in chapter 4. Chapter 5 describes the implementation of the FIR and IIR processor mini-cores based on a discussion of HDL abstraction levels and processor architecture. Also implemented are 2 mini-cores INPUT and OUTPUT facilitating reading/writing test data from/to files. The implementation of the mini-core system communication is described in section 6.1.

The resulting behavioral mini-core system is capable of executing programs constructed for the original mini-core system. It is also possible to create new programs using the assembler for the original system. Small modifications are however necessary to port the programs, these modifications are described in section 5.3.4.

The behavioral mini-core system was tested with an algorithm constructed for the original mini-core system named *filterbank*, test and algorithm are described in chapter 7. The algorithm divides input data into 7 frequency bands using FIR 1 and FIR 2. The algorithm has been modified such that it adds the 7 frequency bands together in IIR 1 to 3, which makes it possible directly to compare input and output data. This is not a complete functional test but is based on the fact, that the principles of both algorithm and mini-core system have been rigorously tested in other projects. The test showed the output data clearly resembling the input data, see figures 7.2 and 7.3. It has not been possible to compare the test results with the original mini-core system.

The behavioral mini-core system is the obvious test platform for expansions of the mini-core system and was therefore employed to serve as a platform

for implementation of methods facilitating easy system level integration of (IP) cores. The implemented methods are based on a general purpose core interface together with the concepts of *communication wrappers* and *core-centric protocols*, which are described in sections 2.1 and 2.2. The choice of implementing OCP as the core-centric protocol was based on a preliminary study, showing it suitable for the mini-core system. Further research has shown the WISHBONE protocol also to fulfill the requirements of the mini-core system. The results of the thesis have however in no way suffered from the choice of implementing OCP. Implementation of OCP in the mini-core system is described in section 6.2.

Also a general purpose interface for the on-chip interconnect facilitating easy substitution of interconnect format is defined and implemented. A behavioral bus module was constructed complying with this interface to ensure correct functionality of the interface, this is described in section 6.3.

The core wrappers and the behavioral bus module have been equipped with the possibility to log information about the data traffic of the system. This data can be post processed providing valuable information about the communication profile of the system. The implementation of the traffic logging is described in section 6.4.

The wrapping of cores and interconnect together with the traffic logging were tested in the behavioral mini-core system with the *filterbank* algorithm. This showed the same functionality as for the original system, except that the behavioral bus module introduced an additional clock cycle to the communication latency. The reason is that unlike the original mini-core system, the bus is now a separate synchronous module. The behavior could be rectified by converting the bus into an asynchronous module.

The major advantages of the wrapper structure implemented are: Increased portability of cores and interconnects ("plug-and-play"), decreased dependency between cores and interconnect resulting in increased design concurrency and decreased workload due to utilization of standard protocols decreasing documentation, supplied with APIs etc.

The major disadvantages are: Overhead from wrapper structure and license limitations due to protocol standards.

The advantages and disadvantages are discussed in detail in section 8.3.

9.1 Future Work

This thesis has taken the basic steps of examining the current methods capable of easing system level integration of (IP) cores in SoCs. However much work is still to be done, especially in determining the disadvantages of these methods. Only assumptions and estimates have been made to address the overhead introduced into the system. Experiments with various systems,

protocols and algorithms must be carried out to provide a clear picture of the possibilities and effects these methods have.

The behavioral mini-core system can serve as a platform for experiments of this sort. The system facilitates easy substitution of cores, interconnect and communication protocol. Experiments currently of most interest are:

- Development and benchmarking of cores and algorithms with more demanding communication format than the FIR and IIR mini-cores.
- An examination of the possibility that an expansion of the OCP or an implementation of another protocol will increase system performance.
- An examination of the overhead introduced into the system by the various expansions, by comparing them to the original communication format, regarding both speed, die area and power consumption. This requires synthesizable implementations to provide precise results.

Bibliography

[1] ARM Limited: *AMBA^{TM} Specification*, Rev. 2.0, 1999.

[2] Peter J. Ashenden: *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers Inc., 1996.

[3] Henry Chang, Larry Cooke, Merril Hunt, Grant Martin, Andrew McNelly and Lee Todd : *Surviving the SOC Revolution*, Kluwer Academic Publishers, 1999.

[4] GNU, www.gnu.org.

[5] Anssi Haverinen, Maxime Leclercq, Norman Weyrich and Drew Wingard: *SystemC^{TM} based SoC Communication Modeling for the OCP^{TM} Protocol*, Version 1.0, www.ocpip.org, October 2002.

[6] IEEE Standard 1149.1-1990: *IEEE Standard Test Access Port and Boundary-Scan Architecture*, 1990.

[7] Niels Christian Arvid Haandbæk: *Design and VLSI Implementation of a Dedicated Low-Power DSP Circuit*, Master's Thesis at IMM, Technical University of Denmark, July 2000.

[8] IBM: *The CoreConnect^{TM} Bus Architecture*, www.chips.ibm.com.

[9] Mogens Isager: *Block Level Interconnect Structures for Low-Power DSP*, Master's Thesis at IMM, Technical University of Denmark, July 2000.

[10] Open Core Protocol International Partnership (OCP-IP): *CoreCreator^{TM}*, www.ocpip.org.

[11] Open Core Protocol International Partnership (OCP-IP): *Open Core Protocol Specification*, Release 1.0, 2001.

[12] Open Core Protocol International Partnership (OCP-IP): *Socket-Centric IP Core Interface Maximizes IP Applications*, www.ocpip.org.

[13] OpenCores, www.opencores.org.

[14] Özgün Paker: *Low Power Audio Signal Processor*, Master's Thesis at IMM, Technical University of Denmark, June 1998.

[15] Özgün Paker: *Low Power Digital Signal Processing*, Ph.D. Thesis at IMM, Technical University of Denmark, June 2002.

[16] David A. Patterson and John L. Hennessy: *Computer Organization & Design - the Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., 2nd edition 1998.

[17] Wade D. Peterson, Silicore Corporation: *Summary of SoC Interconnection Buses*, www.silicore.net, March 2002.

[18] Semiconductor Industry Association: *Summary of the International Technology Roadmap for Semiconductors*, February 2001.

[19] SystemC Language Working Group: *SystemC^{TM} Version 2.0 - User's Guide*, Ver. 2.0, 2001.

[20] SystemC Language Working Group: *Functional Specification for SystemC 2.0*, Ver. 2.0-P, October 2001.

[21] Virtual Socket Interface Alliance (VSIA): *On-Chip Bus Attributes Specification Version 1*, Rev. 2.0, www.vsi.org, September 2001.

[22] Silicore Corporation: *WISHBONE System-on-Chip (SoC) Interconnection Architectures for Portable IP Cores*, Revision B.3, September 2002.

Appendix A

Source Code for Mini-Core Design

This appendix contains all the source code of the behavioral SystemC mini-core model. The modules occur in alphabetized order.

A.1 bus_generic.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : bus_generic.cpp                         **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                 **
 ** Date             : 08.01.2003                              **
 **                                                             **
 ** Function         : Connects all inputs in a bus structure. **
 ** Notes            : Includes bus arbiting, round-robin      **
 **                  : Each core has N channels. They are adressed **
 **                  : consecutivly, i.e. core 0 has 0 to N-1. **
 **                                                             **
 ** Major revisions : 09.01.2003   All communications functioning. **
 **                  : 14.01.2003   Statistics generation added. **
 **                                                             **
 *****************************************************************/

#include "systemc.h"

template< int CORE_COUNT, int DATA_WIDTH, int ADDR_WIDTH, int CHANNEL_WDTH, int
    CHANNEL_CNT, bool STAT_ON >
SC_MODULE(bus_generic) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;                  // active low, synchronous
      reset
  sc_in< sc_uint<DATA_WIDTH> >       data_out[CORE_COUNT];
  sc_out< sc_uint<DATA_WIDTH> >      data_in[CORE_COUNT];
  sc_in< bool >                      req_out[CORE_COUNT];
```

```
sc_out< bool >                          req_in[CORE_COUNT];
sc_in< bool >                           hold_out[CORE_COUNT];
sc_out< bool >                          hold_in[CORE_COUNT];
sc_in< sc_uint<ADDR_WIDTH> >            addr_out[CORE_COUNT];
sc_out<  sc_uint<CHANNEL_WDTH> >        addr_in[CORE_COUNT];


/*********************************
**          Signals              **
*********************************/

int bus_driver;
int hold_driver;
int token_initiator;
int cc_count;
int stat[CORE_COUNT][3];  // stores request and hold counts for cores
FILE *p_report;
FILE *p_stat;
bool holds[CORE_COUNT];


SC_CTOR(bus_generic) {

  SC_METHOD( node_read );
  sensitive_pos << clk;

  SC_METHOD( node_write );
  sensitive_neg << clk;

  SC_METHOD( statistics ); // Creates run-time report and statistics
  sensitive_neg << clk;

  // open files
  char message[] = "Run-time report for simulation of mini-core system.\nShows:\n-Clock
      cycle\n-Node driving bus\n-Hold, node with full buffer stalling bus.\nFor each
      is shown the address of the node responsible for the signal. -1 indicates that no
      node is setting the signal\n\nClock cycle\tNode driving bus\tHold from node\n";

  if (STAT_ON) {
    if ( (p_report = fopen("report_bus.txt", "w")) == NULL)  {
      printf("Error creating report file!!!\n");
      abort();
    }
    fprintf(p_report, message);
  }

} // end constructor


 /*********************************
  ** node_read()                  **
  ** Read data from nodes.        **
  ** Generate hold signal, log    **
  ** bus driver if any.           **
  *********************************/
 void node_read() {

   int index;
   bool req_found;
   bool hold_found;

   if (nrst.read() == 0) {              //active reset
```

```
        token_initiator = 0;                // start with node 0
        for (int i=0; i<CORE_COUNT; i++) {
          hold_in[i].write(0);
          holds[i] = 0;
        }
        bus_driver = -1;
        hold_driver = -1;

      }
      else {
        // defaults
        for (int i=0; i<CORE_COUNT; i++) {
          hold_in[i].write(0);
          holds[i] = 0;
        }
        hold_driver = -1;

        // search for hold from nodes
        hold_found = 0;
        for (int i=0; i<CORE_COUNT; i++) {
          if (hold_out[i].read()) {
            hold_found = 1;
            hold_driver = i;
          }
        }
        // stall cores if necessary
        if (hold_found) {
          for (int i=0; i<CORE_COUNT; i++) {
            hold_in[i].write(1);            // hold all cores ...
            holds[i] = 1;                   // update internal list
          }
          hold_in[hold_driver].write(0);    // ... except driver (to avoid deadlock)
          holds[hold_driver] = 0;
        }

        // look for requests
        req_found = 0;
        for (int i=0; i<CORE_COUNT; i++) {
          index = (token_initiator + i) % CORE_COUNT; // traverse through nodes starting
              with "initiator"
          if(req_out[index] && !holds[index])  {      // request from node not stalled
            if (!req_found) {                         // no previous req, grant permission
              bus_driver = index;                     // grant permission to node
              req_found = 1;
            }
            else {                                    // permission allready granted
              hold_in[index].write(1);                // stall node
            }
          }
        }
        if (!req_found) {     //no bus driver
          bus_driver = -1;
        }

        token_initiator = (token_initiator+1)%CORE_COUNT;  // update token initiator,
            round-robin
      }

    } // END OF node_read
```

```
/*********************************
 ** node_write()                 **
 ** Pass data from bus driver    **
 ** if any.                      **
 *********************************/
void node_write() {

  int dest_channel;
  int local_channel;
  int dest_node;

  if (nrst.read() == 0) {                //active reset
    for (int i=0; i<CORE_COUNT; i++) {
      req_in[i].write(0);
    }
  }
  else {

    for (int i=0; i<CORE_COUNT; i++) {
      req_in[i].write(0);
    }

    if (bus_driver != -1) {              //bus driver exists this CC

      dest_channel = addr_out[bus_driver].read();
      local_channel = dest_channel % CHANNEL_CNT;            // map to local channel
      dest_node = dest_channel / CHANNEL_CNT;                // find target node,
          input core has no input channel
      data_in[dest_node].write(data_out[bus_driver].read()); // set data
      addr_in[dest_node].write(local_channel);               // set channel number
      req_in[dest_node].write(1);
    }
  }
} // END OF node_write

/*********************************
 ** statistics()                 **
 ** Output run-time report and   **
 ** hold and req count to files.**
 *********************************/
void statistics() {

  int i;

  if (nrst == 0) {
    cc_count = 0;
    for (i=0;i<CORE_COUNT;i++) {
        stat[i][0] = 0;
        stat[i][1] = 0;
    }
  }
  else {
    if (STAT_ON) {
      if (bus_driver != -1 || hold_driver != -1) {

        // create report
        fprintf(p_report, "%u\t\t%d\t\t\t%d\n", cc_count, bus_driver, hold_driver);
        fflush(p_report);

        // create statistics
        if ( (p_stat = fopen("stat_bus.txt", "w")) == NULL)  {
          printf("Error creating statistics file!!!\n");
```

```
            abort();
          }
          if (bus_driver != -1)
            stat[bus_driver][0]++;
          for (i=0; i<CORE_COUNT;i++) {
            if (hold_out[i].read())
              stat[i][2]++;
            if (req_out[i].read())
              stat[i][1]++;
          }
          for (i=0; i<CORE_COUNT;i++) {
            fprintf(p_stat,"%u\n%u\n%u\n%u\n",i,stat[i][0],stat[i][1],stat[i][2]);
          }
          fclose(p_stat);
        }

      cc_count++;
      }
    }
  } // END OF statistics

}; // END OF bus_generic
```

A.2 bus_node.cpp

```
/*****************************************************************
 **                                                             **
 ** Name            : bus_node.cpp                              **
 **                                                             **
 ** Author          : Karsten Larsen, c971833                   **
 ** Date            : 19.12.2002                                **
 **                                                             **
 ** Function        : Bus node. Handles bus transaction - including  **
 **                 : local arbiting.                           **
 ** Notes           : Interacts with interface on rising clock edge. **
 **                 : Bus transaction occur on falling clock edge.   **
 **                 : Token passing in initiated each rising clock   **
 **                 : edge and traverses combinatorial through the   **
 **                 : nodes.                                    **
 ** Major revisions : 28.10.2002   First version.               **
 **                 : 29.11.2002   Support of 4 buffers per core     **
 **                 :              implemented.                 **
 **                 : 19.12.2002   Rewritten to fix bugs.        **
 **                                                             **
 *****************************************************************/

#include "systemc.h"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, bool token_initiator >
SC_MODULE(bus_node) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                           clk;
  sc_in< bool >                       nrst;              // active low, synchronous reset
  sc_in< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> >  send;       // packet from interface
  sc_out< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > recv;    // packet to interface
  sc_out< bool >                      send_hold;         // send hold to interface
  sc_in< bool >                       recv_hold;         // receive hold from interface
```

```
    sc_in< bool >                   send_req;           // send request from interface
    sc_out< bool >                  recv_req;           // receive request to interface
    sc_inout_rv<ADDR_WIDTH>         addr;               // address bus
    sc_inout_rv<DATA_WIDTH>         data;               // data bus
    sc_out< bool >                  hold_out;           // hold the entire bus
    sc_in< bool >                   hold_in;            // ...
    sc_in< bool >                   token_in;           // bus permit token input
    sc_out< bool >                  token_out;          // bus permit token output
    sc_out< bool >                  req_out;            // indicate valid data on bus
    sc_out< bool >                  req_in;             // ...

    /*********************************
    **          Signals              **
    *********************************/

    sc_signal<bool> send_ready;
    sc_signal<bool> trig;
    sc_uint<DATA_WIDTH> internal_recv_data;
    sc_uint<CHANNEL_WIDTH> internal_recv_addr;
    sc_uint<DATA_WIDTH> internal_send_data;
    sc_uint<ADDR_WIDTH> internal_send_addr;
    sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> temp;
    sc_lv<ADDR_WIDTH> addr_hi_z;
    sc_lv<DATA_WIDTH> data_hi_z;
    sc_uint<ADDR_WIDTH> temp_addr;

    /*********************************
    **          Variables            **
    *********************************/

    bool token_initiate;
    bool write_to_bus;
    bool recv_ready;
    bool trig_process;
    bool token_int;
    int first_run;
    unsigned int INT_ADDRESS;
    unsigned int INT_ADDRESS_TOP;
    unsigned int int_addr;

    /*********************************
    **         Constructor           **
    *********************************/

    SC_CTOR(bus_node) {

      // create high impedance constants
      int i;
      for (i=0; i<ADDR_WIDTH; i++)
        addr_hi_z[i] = 'Z';
      for (i=0; i<DATA_WIDTH; i++)
        data_hi_z[i] = 'Z';

      token_initiate = token_initiator;  // indicates whether node is to initiate token
          passing

      SC_METHOD( node_read );
      sensitive_pos << clk;

      SC_METHOD( node_write );
      sensitive_neg << clk;
```

```
    SC_METHOD( token_handle );
    sensitive << token_in;

    SC_METHOD( token_init );
    sensitive_pos << clk;

    SC_METHOD( token_rst );
    sensitive_neg << clk;

  } // end constructor


  /*********************************
   **        node_read()          **
   ** Reads on positive clk edge  **
   *********************************/
  void node_read() {

    if (nrst.read() == 0) {  //active reset
      INT_ADDRESS = 4*ADDRESS;
      INT_ADDRESS_TOP = INT_ADDRESS+CHANNEL_COUNT-1;
      recv_ready = 0;
      send_ready = 0;
    }
    else {                                                          // positive clockedge
      // receive - from network to interface
      if (req_in) {                                                 // valid data on bus
        temp_addr = addr.read();                                    // ...
        int_addr = temp_addr;                                       // type conversion for comparison
        if (int_addr >= INT_ADDRESS && int_addr <= INT_ADDRESS_TOP) { //data for this node
          internal_recv_data = data.read();
          internal_recv_addr = addr.read();
          internal_recv_addr -= INT_ADDRESS;                        // map into internal core channel (0-3)
          recv_ready = 1;                                           // receive data ready
        }
      }
      else {

      }

      //send - from interface to network
      if (send_req) {                                                                          // send from interface
        internal_send_data = (send.read()).range(DATA_WIDTH-1, 0);
        internal_send_addr = (send.read()).range(ADDR_WIDTH+DATA_WIDTH-1, DATA_WIDTH);
        send_ready = 1;                                                                        // send data ready
      }
    }
    return;
  } // END OF node_read()


  /*********************************
   **        node_write()         **
   ** Writes on negative clk edge **
```

```
 ******************************/
void node_write() {

  if (nrst.read() == 0) {              //active reset
    hold_out.write(0);
    send_hold.write(0);
    addr.write(addr_hi_z);
    data.write(data_hi_z);
    token_initiate = token_initiator;
  }

  else {

    hold_out.write(recv_hold.read());              // pass receive hold from interface
        to network

    // receive - from network to interface
    recv_req.write(0);                             // default value
    if (recv_ready) {                              // data from bus to interface
        waiting
      if (!recv_hold.read()) {                     // no hold from interface
        recv_req.write(1);
        temp[CHANNEL_WIDTH+DATA_WIDTH] = '1';
        temp.range(CHANNEL_WIDTH+DATA_WIDTH-1, DATA_WIDTH) = internal_recv_addr;
        temp.range(DATA_WIDTH-1, 0) = internal_recv_data;
        recv.write(temp);
        recv_ready = 0;
      }
    }


    // send - from interface to network

    // defaults:
    send_hold.write(0);
    addr.write(addr_hi_z);
    data.write(data_hi_z);
    req_out.write(0);

    if (hold_in.read()) {                // hold from network
      send_hold.write(1);                // stall interface
    }
    else {                               // no hold
      if (write_to_bus) {                // granted bus permission
        req_out.write(1);                // bus data valid
        data.write(internal_send_data);
        addr.write(internal_send_addr);
        send_ready = 0;
      }

      if (send_ready && !write_to_bus) {
        send_hold.write(1);
      }
    }
  }
  return;
} // END OF node_write()


/*******************************
 **       token_handle()        **
 ** Handles token passing       **
```

```
 ** Combinatorial                **
 *********************************/
void token_handle() {

  if (clk) {                      // Only active when clock is high
    write_to_bus = 0;

    if (token_initiate) {         // token initiator, token cycle complete
      if (token_in) {             // received token
        token_initiate = 1;       // token initiator, next clockcycle
        if (send_ready) {         // waiting to transmit data
          write_to_bus = 1;       // ok to write to bus
        }
      }
      else {
        token_initiate = 0;       // new token initiator
      }
    }

    else {                        // not token initiator
      if (token_in) {             // received token

        if (send_ready) {         // waiting to transmit data
          token_out.write(0);     // do not pass token
          write_to_bus = 1;       // ok to write to bus
          token_initiate = 1;     // token initiator, next clockcycle
        }
        else {                    // no data
          token_out.write(1);     // no data, pass token
          token_initiate = 0;
        }
      }
      else {                      // no token
        token_out.write(0);
        token_initiate = 0;
      }
    }
  }
  return;
} // END OF token_handle()


/*********************************
 **        token_init()          **
 ** Initiates token chain,       **
 ** if told so                   **
 *********************************/
void token_init (){

  if (nrst.read() == 0) {          //active reset
    write_to_bus = 0;
    token_out.write(0);
    first_run = 2;
  }
  else {
    if (token_initiate) {          // token initiator
      token_out.write(1);          // pass token
      write_to_bus = 0;
    }
  }
  return;
} // END OF token_init()
```

```
  void token_rst() {

    // set token_initate if 0 on input a falling clockedge
    if (nrst.read() == 1) {
      if (first_run == 0) {
        if (token_initiate) {
          if (token_in.read() == 0) {
            token_initiate = 0;
          }
        }
      }
      else {
        first_run--;
      }

      // reset token ring
      token_out.write(0);
    }
  } // END OF token_rst()

};
```

A.3 core_system.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : core_system.cpp                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 08.01.2003                                   **
 **                                                                 **
 ** Function         : Connects wrapped cores and nodes with network. **
 **                                                                 **
 ** Major revisions : 08.01.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "declarations.h"
#include "fir_core_node.cpp"
#include "iir_core_node.cpp"
#include "input_core_node.cpp"
#include "output_core_node.cpp"
#include "bus_generic.cpp"

template< int FIR_COUNT, int IIR_COUNT, int DATA_WIDTH, int ADDR_WIDTH, const char *
    imem_name1, const char *cmem_name1, const char *fmem_name1, const char *imem_name2,
    const char *cmem_name2, const char *fmem_name2, const char *imem_name3, const char *
    cmem_name3, const char *fmem_name3, const char *imem_name4, const char *cmem_name4,
    const char *imem_name5, const char *cmem_name5, const char *imem_name6, const char *
    cmem_name6, const char *input_name, const char *output_name, bool STATISTICS_ON,
    const char *input_stat_name, const char *output_stat_name, const char *fir1_stat_name
    , const char *fir2_stat_name, const char *fir3_stat_name, const char *iir1_stat_name
    , const char *iir2_stat_name, const char *iir3_stat_name>
SC_MODULE(core_system) {
```

```
    /*********************************
    **          Ports                **
30  *********************************/

    sc_in_clk                         clk;
    sc_in< bool >                     nrst;             // active low, synchronous reset

    /*********************************
    **          Signals              **
    *********************************/

    sc_signal< sc_uint<DATA_WIDTH> > data_out[FIR_COUNT+IIR_COUNT+2];
40  sc_signal< sc_uint<DATA_WIDTH> > data_in[FIR_COUNT+IIR_COUNT+2];
    sc_signal< bool > req_out[FIR_COUNT+IIR_COUNT+2];
    sc_signal< bool > req_in[FIR_COUNT+IIR_COUNT+2];
    sc_signal< bool > hold_out[FIR_COUNT+IIR_COUNT+2];
    sc_signal< bool > hold_in[FIR_COUNT+IIR_COUNT+2];
    sc_signal< sc_uint<ADDR_WIDTH> > addr_out[FIR_COUNT+IIR_COUNT+2];
    sc_signal< sc_uint<CHANNEL_WIDTH> > addr_in[FIR_COUNT+IIR_COUNT+2];


    /*********************************
50  **          Components           **
    *********************************/

    input_core_node<7, DATA_WIDTH, ADDR_WIDTH, input_name, STATISTICS_ON, input_stat_name
        > *input_core_node1;
    output_core_node<6, DATA_WIDTH, ADDR_WIDTH, output_name, STATISTICS_ON,
        output_stat_name> *output_core_node1;

    fir_core_node<0, DATA_WIDTH, ADDR_WIDTH, imem_name1, cmem_name1, fmem_name1,
        STATISTICS_ON, fir1_stat_name> *fir_core_node1;
    fir_core_node<1, DATA_WIDTH, ADDR_WIDTH, imem_name2, cmem_name2, fmem_name2,
        STATISTICS_ON, fir2_stat_name> *fir_core_node2;
    fir_core_node<2, DATA_WIDTH, ADDR_WIDTH, imem_name3, cmem_name3, fmem_name3,
        STATISTICS_ON, fir3_stat_name> *fir_core_node3;
    iir_core_node<3, DATA_WIDTH, ADDR_WIDTH, imem_name4, cmem_name4, STATISTICS_ON,
        iir1_stat_name> *iir_core_node1;
60  iir_core_node<4, DATA_WIDTH, ADDR_WIDTH, imem_name5, cmem_name5, STATISTICS_ON,
        iir2_stat_name> *iir_core_node2;
    iir_core_node<5, DATA_WIDTH, ADDR_WIDTH, imem_name6, cmem_name6, STATISTICS_ON,
        iir3_stat_name> *iir_core_node3;

    bus_generic <FIR_COUNT+IIR_COUNT+2, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH,
        CHANNEL_COUNT, STATISTICS_ON> *bus;

    /*********************************
    **        Constructor            **
    *********************************/

    SC_CTOR(core_system) {
70

      // input core

      input_core_node1 = new input_core_node<7, DATA_WIDTH, ADDR_WIDTH, input_name,
          STATISTICS_ON, input_stat_name> ("input_core_node1");;
      input_core_node1->clk(clk);
      input_core_node1->nrst(nrst);
      input_core_node1->data_out(data_out[7]);
      input_core_node1->data_in(data_in[7]);
      input_core_node1->req_out(req_out[7]);
      input_core_node1->req_in(req_in[7]);
```

```
80      input_core_node1->addr_out(addr_out[7]);
        input_core_node1->addr_in(addr_in[7]);
        input_core_node1->hold_out(hold_out[7]);
        input_core_node1->hold_in(hold_in[7]);


        // output core

        output_core_node1 = new output_core_node<6, DATA_WIDTH, ADDR_WIDTH, output_name,
            STATISTICS_ON, output_stat_name> ("output_core_node1");;
        output_core_node1->clk(clk);
90      output_core_node1->nrst(nrst);
        output_core_node1->data_out(data_out[6]);
        output_core_node1->data_in(data_in[6]);
        output_core_node1->req_out(req_out[6]);
        output_core_node1->req_in(req_in[6]);
        output_core_node1->addr_out(addr_out[6]);
        output_core_node1->addr_in(addr_in[6]);
        output_core_node1->hold_out(hold_out[6]);
        output_core_node1->hold_in(hold_in[6]);

100

        // FIR cores

        fir_core_node1 = new fir_core_node<0, DATA_WIDTH, ADDR_WIDTH, imem_name1, cmem_name1
            , fmem_name1, STATISTICS_ON, fir1_stat_name> ("fir_core_node1");
        fir_core_node1->clk(clk);
        fir_core_node1->nrst(nrst);
        fir_core_node1->data_out(data_out[0]);
        fir_core_node1->data_in(data_in[0]);
        fir_core_node1->req_out(req_out[0]);
        fir_core_node1->req_in(req_in[0]);
110     fir_core_node1->addr_out(addr_out[0]);
        fir_core_node1->addr_in(addr_in[0]);
        fir_core_node1->hold_out(hold_out[0]);
        fir_core_node1->hold_in(hold_in[0]);

        fir_core_node2 = new fir_core_node<1, DATA_WIDTH, ADDR_WIDTH, imem_name2, cmem_name2
            , fmem_name2, STATISTICS_ON, fir2_stat_name> ("fir_core_node2");
        fir_core_node2->clk(clk);
        fir_core_node2->nrst(nrst);
        fir_core_node2->data_out(data_out[1]);
        fir_core_node2->data_in(data_in[1]);
120     fir_core_node2->req_out(req_out[1]);
        fir_core_node2->req_in(req_in[1]);
        fir_core_node2->addr_out(addr_out[1]);
        fir_core_node2->addr_in(addr_in[1]);
        fir_core_node2->hold_out(hold_out[1]);
        fir_core_node2->hold_in(hold_in[1]);

        fir_core_node3 = new fir_core_node<2, DATA_WIDTH, ADDR_WIDTH, imem_name3, cmem_name3
            , fmem_name3, STATISTICS_ON, fir3_stat_name> ("fir_core_node3");
        fir_core_node3->clk(clk);
        fir_core_node3->nrst(nrst);
130     fir_core_node3->data_out(data_out[2]);
        fir_core_node3->data_in(data_in[2]);
        fir_core_node3->req_out(req_out[2]);
        fir_core_node3->req_in(req_in[2]);
        fir_core_node3->addr_out(addr_out[2]);
        fir_core_node3->addr_in(addr_in[2]);
        fir_core_node3->hold_out(hold_out[2]);
        fir_core_node3->hold_in(hold_in[2]);
```

```
140     // IIR cores

        iir_core_node1 = new iir_core_node<3, DATA_WIDTH, ADDR_WIDTH, imem_name4, cmem_name4
            , STATISTICS_ON, iir1_stat_name> ("iir_core_node1");
        iir_core_node1->clk(clk);
        iir_core_node1->nrst(nrst);
        iir_core_node1->data_out(data_out[3]);
        iir_core_node1->data_in(data_in[3]);
        iir_core_node1->req_out(req_out[3]);
        iir_core_node1->req_in(req_in[3]);
        iir_core_node1->addr_out(addr_out[3]);
150     iir_core_node1->addr_in(addr_in[3]);
        iir_core_node1->hold_out(hold_out[3]);
        iir_core_node1->hold_in(hold_in[3]);

        iir_core_node2 = new iir_core_node<4, DATA_WIDTH, ADDR_WIDTH, imem_name5, cmem_name5
            , STATISTICS_ON, iir2_stat_name> ("iir_core_node2");
        iir_core_node2->clk(clk);
        iir_core_node2->nrst(nrst);
        iir_core_node2->data_out(data_out[4]);
        iir_core_node2->data_in(data_in[4]);
        iir_core_node2->req_out(req_out[4]);
160     iir_core_node2->req_in(req_in[4]);
        iir_core_node2->addr_out(addr_out[4]);
        iir_core_node2->addr_in(addr_in[4]);
        iir_core_node2->hold_out(hold_out[4]);
        iir_core_node2->hold_in(hold_in[4]);

        iir_core_node3 = new iir_core_node<5, DATA_WIDTH, ADDR_WIDTH, imem_name6, cmem_name6
            , STATISTICS_ON, iir3_stat_name> ("iir_core_node3");
        iir_core_node3->clk(clk);
        iir_core_node3->nrst(nrst);
        iir_core_node3->data_out(data_out[5]);
170     iir_core_node3->data_in(data_in[5]);
        iir_core_node3->req_out(req_out[5]);
        iir_core_node3->req_in(req_in[5]);
        iir_core_node3->addr_out(addr_out[5]);
        iir_core_node3->addr_in(addr_in[5]);
        iir_core_node3->hold_out(hold_out[5]);
        iir_core_node3->hold_in(hold_in[5]);

        // network, this is interchangeable

180     bus = new bus_generic<FIR_COUNT+IIR_COUNT+2,DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH,
            CHANNEL_COUNT, 1> ("bus");
        bus->clk(clk);
        bus->nrst(nrst);
        for (int i=0; i<FIR_COUNT+IIR_COUNT+2; i++) {
          bus->data_out[i](data_out[i]);
          bus->data_in[i](data_in[i]);
          bus->req_out[i](req_out[i]);
          bus->req_in[i](req_in[i]);
          bus->addr_out[i](addr_out[i]);
          bus->addr_in[i](addr_in[i]);
190       bus->hold_out[i](hold_out[i]);
          bus->hold_in[i](hold_in[i]);
        }

      } // end constructor
```

```
}; // END OF core_system
```

A.4 declarations.h

```
/*********************************************************************
 **                                                                 **
 ** Name             : declarations.h                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                     **
 ** Date             : 02.01.2003                                  **
 **                                                                 **
 ** Function         : Constants for mini-core.                    **
 ** Major revisions : 02.01.2003   Final version                   **
 **                                                                 **
 *********************************************************************/

/*********
 ** FIR **
 *********/

// Memory sizes
#define IMEM_SIZE     50
#define CMEM_SIZE     100
#define FMEM_SIZE     10
#define DELAY_LENGTH 300
#define FILTER_WIDTH 16

// Misc. constant

#define IMM_WIDTH          8
#define COEFF_WIDTH        16  // width of fixed point coefficients
#define COEFF_INT_WIDTH    1   // bits before decimal point
#define SAMPLE_WIDTH       16  // width of fixed point samples
#define SAMPLE_INT_WIDTH   1   // bits before decimal point

// Indexes into filter memory

#define FM_LN       0
#define FM_BCP      1
#define FM_CP       2
#define FM_BDP      3
#define FM_DP1      4
#define FM_DP2      5

// Opcodes - as in original design

#define OP_NOP       0
#define OP_MACC      1
#define OP_ASMACC    2
#define OP_ADDI      3
#define OP_SUBI      4
#define OP_LSET      5
#define OP_HSET      6
#define OP_SWITCH    7
#define OP_LOAD      8
#define OP_STORE     9
#define OP_SEND      10
#define OP_RECEIVE   11
#define OP_JMP       12
#define OP_MOVREG    14
```

```
#define OP_BRA        15

// Flags - as in original design
// Constants indicate bit position or value

#define FLAG_CLR        3 // bit position
#define FLAG_FIN        1 // ..
#define FLAG_SUB        0 // ..
#define FLAG_ZERO       3 // ..
#define FLAG_NZERO      3 // ..
#define FLAG_MOD        3 // bit position
#define FLAG_CP         0 // value
#define FLAG_DP1        1 // bit position
#define FLAG_DP2        2 // ..
#define FLAG_MACC       3 // value
#define FLAG_CLR        3 // bit position
#define FLAG_COMP       4 // ..
#define FLAG_TMP1       5 // value
#define FLAG_TMP2       6 // ..
#define FLAG_TMP3       7 // ..
#define FLAG_DM         3 // bit position
#define FLAG_CM         3 // ..
#define FLAG_REGDM      3 // ..
#define FLAG_DMREG      3 // ..

/*********
 ** IIR **
 *********/

#define INSTRUCTION_WIDTH 13
#define IMM_WIDTH_IIR 3
#define READ_WIDTH 2
#define WRITE_WIDTH 6
#define RX_WIDTH 2
#define RY_WIDTH 2
#define TARGET_ADDRESS_WIDTH 5
#define EXTRA_PRECISION 4
#define IMSIZE 502
#define CMSIZE 50
#define COEFF_LENGTH 8
#define INSTRUCTION_LENGTH 16
#define BIQUAD_WIDTH 1
#define DATA_INT_WIDTH 1
#define REGISTER_COUNT 4


// Opcodes - as in original design

#define OP_NOP_IIR      0
#define OP_ADD_IIR      1
#define OP_SUB_IIR      2
#define OP_MOV_IIR      3
#define OP_SHFL_IIR     11
#define OP_SHFR_IIR     7
#define OP_BIQ_IIR      4
#define OP_JUMP_IIR     14
#define OP_SEND_IIR     16
#define OP_RECEIVE_IIR  15

/***************
 ** Interface **
```

```
120   **************/

#define CHANNEL_WIDTH 2
#define CHANNEL_COUNT 4
```

A.5 fir_core.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : fir_core.cpp                                 **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 27.11.2002                                   **
 **                                                                 **
 ** Function         : Behavioral mini-core implementation of a     **
 **                  : dedicated FIR processor.                     **
10 ** Notes            : Communicates with external components by send **
 **                  : and receive instructions.                    **
 **                  : Memories are preloaded from files.           **
 ** Major revisions : 15.10.2002   Complete FIR except communication. **
 **                  : 01.11.2002   Communication implemented.      **
 **                  : 29.11.2002   4 channels per core added.      **
 **                                                                 **
 *********************************************************************/


20 #define SC_INCLUDE_FX

#include "systemc.h"
#include "declarations.h"


template<int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char *
    cmem_name, const char *fmem_name>
SC_MODULE(fir_core) {

  /*********************************
30 **          Ports                **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;        // active low, synchronous reset
  sc_out< bool >                     req;         // request to interface
  sc_out< bool >                     rw;          // send or receive (1=rec) to interface
  sc_in< bool >                      hold;        // hold from interface
  sc_out< sc_uint<DATA_WIDTH> >      data_out;    // data to interface
  sc_out< sc_uint<ADDR_WIDTH> >      write_addr;  // send address to interface
40 sc_out< sc_uint<CHANNEL_WIDTH> > read_addr;   // read channel to interface
  sc_in< sc_uint<DATA_WIDTH> >       data_in;     // data from interface


  /*********************************
  **    Signals and Registers     **
  *********************************/

  int PC;                                                          // program
      counter
  sc_uint< 8+IMM_WIDTH > instruction;                              // opcode(4):
      flags(4):immediate(IMM_WIDTH)
50 sc_uint< IMM_WIDTH > immediate_usig;
  sc_int< IMM_WIDTH > immediate_sig;
```

```
    sc_int<IMM_WIDTH+4> jump_immediate;
    sc_uint<4> flags;
    sc_uint<4> opcode;
    sc_int< DATA_WIDTH > macc_reg;                                          // macc register
    sc_int< DATA_WIDTH > comp_reg;                                          // complementary
        output register
    sc_int< DATA_WIDTH > tmp_reg[3];                                        // general
        purpose registers 1-3
    sc_fixed< DATA_WIDTH,SAMPLE_INT_WIDTH,SC_RND,SC_SAT > macc_reg_fix;     // fixed point
        macc register
    sc_fixed< DATA_WIDTH,SAMPLE_INT_WIDTH,SC_RND,SC_SAT > comp_reg_fix;     // fixed point
        comp register
    sc_fixed< COEFF_WIDTH,COEFF_INT_WIDTH,SC_RND,SC_SAT > coeff_fix;        // fixed point
        coefficient
    sc_fixed< COEFF_WIDTH,COEFF_INT_WIDTH,SC_RND,SC_SAT > delayline1_fix;   // ...
    sc_fixed< COEFF_WIDTH,COEFF_INT_WIDTH,SC_RND,SC_SAT > delayline2_fix;   // fixed point
        delay-line values
    sc_bit reg_nfilter;                 // indicates dest/source register type
    bool req_send;                      // indicates whether a request has been send in a
        recieve instruction
    int sr_count;                       // used in send and receive
    int reg;
    int imm;

    //temp signals
    sc_uint< FILTER_WIDTH > temp_uint;

    /*********************************
    **          Constants           **
    *********************************/

    sc_uint<1> ZERO_UINT;
    sc_uint<1> ONE_UINT;
    sc_uint<8> DELAY_LENGTH_UINT;


    /*********************************
    **          Memories            **
    *********************************/

    sc_uint<8+IMM_WIDTH> imem[IMEM_SIZE];       // instruction memory
    sc_uint<FILTER_WIDTH> fmem[FMEM_SIZE][6];   // filter memory
    sc_uint<COEFF_WIDTH> cmem[CMEM_SIZE];       // filter coefficient memory
    sc_uint<DATA_WIDTH> dmem[DELAY_LENGTH];     // delay-line memory


    /*********************************
    **          Pointers            **
    *********************************/

    int curr;                           // Filter pointer
    sc_int<DATA_WIDTH> *reg_ptr;        // ...
    sc_uint<FILTER_WIDTH> *reg_fm_ptr;  // pointers to destination/source register

    /*********************************
    **            Files             **
    *********************************/

    FILE *p_ifile;
    FILE *p_cfile;
    FILE *p_ffile;
```

```
/*********************************
**            Debug              **
*********************************/

int macc_int;
int delayline_int1;
int delayline_int2;
int coeff_int;
int reg_ptr_int;
int dp1, dp2, cp;
int filt_ln;
int temp;

/*********************************
**        Constructor            **
*********************************/

SC_CTOR( fir_core ) {

/*********************************
**   Load memories from files    **
*********************************/

  printf("Loading mems from file...\n");

  load_imem();
  load_cmem();
  load_fmem();

  printf("All files loaded.\n");


  // methods
  SC_METHOD( read_inst );
  sensitive_pos << clk;

}


/*********************************
**          read_inst()          **
**        read instruction       **
*********************************/

void read_inst() {

  if (nrst.read() == 0) {
    PC = 0;
    req_send = 0;
  }

  else {  // clockedge
    if (!hold.read()) {  // no hold from interface
      instruction = imem[PC];
      control();        // decode and perform operation
    }
    else {
      req.write(1);
    }
  }
  return;
}
```

```
170
    /*********************************
    **          control()            **
    **  mux instruction and perform **
    **  instruction.                 **
    *********************************/

    void control() {

      int i;
180
      if (nrst.read() == 0) {  // reset active

        ZERO_UINT = 0;
        ONE_UINT = 1;
        DELAY_LENGTH_UINT = DELAY_LENGTH;
        curr = 0;
        req.write(0);
        sr_count = 0;
      }
190
      else {  // no reset

        req.write(0);   // default value

        immediate_usig = instruction.range(IMM_WIDTH-1, 0);    // unsigned
        immediate_sig = instruction.range(IMM_WIDTH-1, 0);     // signed
        jump_immediate = instruction.range(IMM_WIDTH+3, 0);    // signed
        flags = instruction.range(IMM_WIDTH+3, IMM_WIDTH);
        opcode = instruction.range(IMM_WIDTH+7, IMM_WIDTH+4);
200
        // type conversions for printf
        reg = flags.range(2,0);
        imm = immediate_usig ;


        //determine source/destination register (if any)
        switch(reg) {

        case FLAG_CP:
210       reg_nfilter = '0';
          reg_fm_ptr = &(fmem[curr][FM_CP]);
          break;
        case FLAG_DP1:
          reg_nfilter = '0';
          reg_fm_ptr = &(fmem[curr][FM_DP1]);
          break;
        case FLAG_DP2:
          reg_nfilter = '0';
          reg_fm_ptr = &(fmem[curr][FM_DP2]);
220       break;
        case FLAG_MACC:
          reg_nfilter = '1';
          reg_ptr = &macc_reg;
          break;
        case FLAG_COMP:
          reg_nfilter = '1';
          reg_ptr = &comp_reg;
          break;
        case FLAG_TMP1:
230       reg_nfilter = '1';
```

```
        reg_ptr = &tmp_reg[0];
        break;
      case FLAG_TMP2:
        reg_nfilter = '1';
        reg_ptr = &tmp_reg[1];
        break;
      case FLAG_TMP3:
        reg_nfilter = '1';
        reg_ptr = &tmp_reg[2];
        break;
      }

      // decode and execute instruction
      switch(opcode) {  // determine instruction type


      /*********************************
      **       multiply-accumulate      **
      *********************************/
      case OP_MACC:

        if (flags[FLAG_CLR]) {  // clear register
          macc_reg = 0;
          macc_reg_fix = 0;
        }

        // conversions from integer(bitvector) to fixed point
        for (i=0; i<DATA_WIDTH; i++) {
          macc_reg_fix[i] = macc_reg[i];
          delayline1_fix[i] = (dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ])[i];
        }
        for (i=0; i<COEFF_WIDTH; i++)
          coeff_fix[i] = (cmem[ fmem[curr][FM_BCP]+fmem[curr][FM_CP] ])[i];

        macc_reg_fix += coeff_fix * delayline1_fix;

        // update macc register
        for (i=0; i<DATA_WIDTH; i++)
          macc_reg[i] = macc_reg_fix[i];

        // complementary
        if (flags[2]) { // complementary output
          for (i=0; i<DATA_WIDTH; i++)
            comp_reg_fix[i] = comp_reg[i];
          comp_reg_fix = delayline1_fix - macc_reg_fix;
          for (i=0; i<DATA_WIDTH; i++)
            comp_reg[i] = comp_reg_fix[i];
        }

        fmem[curr][FM_DP1] = (fmem[curr][FM_DP1]+immediate_usig) % fmem[curr][FM_LN];

        if (flags[FLAG_FIN]) {   // final computation
          fmem[curr][FM_DP2] = (fmem[curr][FM_DP2]+immediate_usig-ONE_UINT) % fmem[curr][
              FM_LN];
        }

        //update coefficient pointer
        fmem[curr][FM_CP]++;

        break;
```

```
        /**********************************
        ** add/subtract-multiply-accumulate **
        **********************************/
        case OP_ASMACC:

          if (flags[FLAG_CLR])
            macc_reg = 0;

300       delayline_int1 = dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ];
          delayline_int2 = dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP2] ];
          coeff_int = cmem[ fmem[curr][FM_BCP]+fmem[curr][FM_CP] ];

          // conversions from integer(bitvector) to fixed point
          for (i=0; i<DATA_WIDTH; i++) {
            macc_reg_fix[i] = macc_reg[i];
            delayline1_fix[i] = (dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ])[i];
            delayline2_fix[i] = (dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP2] ])[i];
          }
310
          for (i=0; i<COEFF_WIDTH; i++)
            coeff_fix[i] = (cmem[ fmem[curr][FM_BCP]+fmem[curr][FM_CP] ])[i];

          if (flags[FLAG_SUB])    // subtract
            macc_reg_fix += coeff_fix * ( delayline1_fix - delayline2_fix );
          else                    // add
            macc_reg_fix += coeff_fix * ( delayline1_fix + delayline2_fix );

          // update pointers
320       fmem[curr][FM_CP]++;
          fmem[curr][FM_DP1] = (fmem[curr][FM_DP1] + immediate_usig) % fmem[curr][FM_LN];

          if ( flags[FLAG_FIN] ) {          // last calculation
            fmem[curr][FM_DP2] = (fmem[curr][FM_DP2] + immediate_usig - ONE_UINT) % fmem[
                curr][FM_LN];
          }
          else {                            // not last calculation
            fmem[curr][FM_DP2] = (fmem[curr][FM_DP2] - immediate_usig) % fmem[curr][FM_LN];
          }

330       // update macc register
          for (i=0; i<DATA_WIDTH; i++)
            macc_reg[i] = macc_reg_fix[i];

          // debug
          macc_int = macc_reg;

          break;


340     /*********************************
        **          add-immediate        **
        *********************************/
        case OP_ADDI:

          if (reg_nfilter){
            *reg_ptr += immediate_sig;
            if ( flags[FLAG_MOD] && *reg_ptr > fmem[curr][FM_LN] )
              *reg_ptr -= fmem[curr][FM_LN];                          // wrap around
          }
350       else {
            *reg_fm_ptr += immediate_sig;
            if ( flags[FLAG_MOD] && *reg_fm_ptr > fmem[curr][FM_LN] )
```

```
        *reg_fm_ptr -= fmem[curr][FM_LN];                    // wrap around
      }

      break;


    /*********************************
    **          sub-immediate         **
    *********************************/
    case OP_SUBI:

      if (reg_nfilter){
        *reg_ptr -= immediate_sig;
        if ( flags[FLAG_MOD] && *reg_ptr < ZERO_UINT )
          *reg_ptr += fmem[curr][FM_LN];                     // wrap around
      }
      else {
        *reg_fm_ptr -= immediate_sig;
        if ( flags[FLAG_MOD] && *reg_fm_ptr < ZERO_UINT )
          *reg_fm_ptr += fmem[curr][FM_LN];                  // wrap around
      }

      break;


    /*********************************
    **          set low-part          **
    *********************************/
    case OP_LSET:

      if (reg_nfilter){
        (*reg_ptr).range(IMM_WIDTH-1, 0) = immediate_usig;
        (*reg_ptr).range(DATA_WIDTH-1, IMM_WIDTH) = 0;
      }
      else {
        (*reg_fm_ptr).range(IMM_WIDTH-1, 0) = immediate_usig;
        (*reg_fm_ptr).range(FILTER_WIDTH-1, IMM_WIDTH) = 0;
      }

      break;

    /*********************************
    **         set high-part          **
    *********************************/
    case OP_HSET:

      if (reg_nfilter){
        (*reg_ptr).range(DATA_WIDTH-1, DATA_WIDTH-IMM_WIDTH) = immediate_usig;
        (*reg_ptr).range(DATA_WIDTH-IMM_WIDTH-1, 0) = 0;
      }
      else {
        (*reg_fm_ptr).range(FILTER_WIDTH-1, FILTER_WIDTH-IMM_WIDTH) = immediate_usig;
        (*reg_fm_ptr).range(FILTER_WIDTH-IMM_WIDTH-1, 0) = 0;
      }

      break;


    /*********************************
    **          switch filter         **
    *********************************/
    case OP_SWITCH:
```

```
    curr = immediate_usig;

    if (flags[FLAG_CLR])
      fmem[curr][FM_CP] = 0;
420
    break;


  /*******************************
  **   memory-register-transfer   **
  *******************************/
  case OP_MOVREG:

    if (flags[FLAG_REGDM]) {        // FROM register TO delay-line
430   if (reg_nfilter){
        dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ] = *reg_ptr;
        temp = *reg_ptr;
      }
      else {
        dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ] = *reg_fm_ptr;
        temp = *reg_ptr;
      }
    }
    else {                          // FROM delay-line TO register
440   if (reg_nfilter){
        temp = dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ];
        *reg_ptr = dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ];
      }
      else {
        temp = dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ];
        *reg_fm_ptr=dmem[ fmem[curr][FM_BDP]+fmem[curr][FM_DP1] ];
      }
    }

450 break;


  /*******************************
  **      load into register      **
  *******************************/
  case OP_LOAD:


    if (flags[FLAG_DM]) {       // load from delay-line
460   if (reg_nfilter){
        *reg_ptr = dmem[immediate_usig];
      }
      else {
        *reg_fm_ptr = dmem[immediate_usig];
      }
    }
    else {                      // load from coefficient memory
      if (reg_nfilter){
        *reg_ptr = cmem[immediate_usig];
470   }
      else {
        *reg_fm_ptr = cmem[immediate_usig];
      }
    }

    break;
```

```
        /*********************************
        **        store in memory        **
        *********************************/
        case OP_STORE:

          if (flags[FLAG_DM]) {       // load from delay-line
            if (reg_nfilter){
              dmem[immediate_usig] = *reg_ptr;
            }
            else {
              dmem[immediate_usig] = *reg_fm_ptr;
            }
          }
          else {                      // load from coefficient memory
            if (reg_nfilter){
              cmem[immediate_usig] = *reg_ptr;
            }
            else {
              cmem[immediate_usig] = *reg_fm_ptr;
            }
          }

          break;


        /*********************************
        **             branch            **
        *********************************/
        case OP_BRA:
          if (!flags[FLAG_ZERO] && tmp_reg[reg] == ZERO_UINT)    // branch reg == 0
            PC += immediate_sig;
          else
            if (flags[FLAG_NZERO] && tmp_reg[reg] != ZERO_UINT) // branch reg != 0
              PC += immediate_sig;

          break;


        /*********************************
        **              jump             **
        *********************************/
        case OP_JMP:

          PC = jump_immediate;
          PC--;

          break;


        /*********************************
        **              send             **
        *********************************/
        case OP_SEND:

          send(reg,imm);

          break;


        /*********************************
```

```
      **              receive             **
      *********************************/
      case OP_RECEIVE:

        receive(reg,imm);

        break;


      /*********************************
      **                nop              **
      *********************************/
      case OP_NOP:

        break;

      default:

        printf("No such FIR(%u) operation!\n", ADDRESS);
        break;  //no such operation
      }

    PC++;
  }

  return;
}


/*********************************
 **         send(reg,imm)         **
 ** Sends contents of reg to     **
 ** output channel indicated by **
 ** imm.                         **
 *********************************/

void send(int reg, int channel) {

  int value;


  if (sr_count == 0) {
    if (reg_nfilter){
      value = *reg_ptr;
    }
    else {
      value = *reg_fm_ptr;
    }

    // pass send request to interface
    write_addr.write(channel);
    data_out.write(value);
    req.write(1);
    rw.write(0);

    PC--;
    sr_count++;

    //debugging
    printf("FIR(%u) sending... \t register: %u \t value: %x \t to channel: %u \n",
        ADDRESS, reg, value, channel);
  }
```

```
600       else {
            req.write(0);
            sr_count = 0;
          }
          return;
        }



        /*******************************
         **       receive(reg,imm)       **
610      ** Recieves data package from  **
         ** channel indicated by imm    **
         ** and stores it in register   **
         ** indicated by reg.           **
         *******************************/

        void receive(int reg, int channel) {

          int value;  //debug
620
          if (sr_count == 0) { // 1st cycle of receive
            // send request to interface
            req.write(1);
            rw.write(1);
            read_addr.write(channel); // channel to read from
            // make sure 2nd cycle is performed
            PC--;
            sr_count++;
          }
630
          else {                // 2nd cycle
            if (sr_count == 1) {
              PC--;
              sr_count++;
              req.write(0);
            }

            else {              // 3rd and last cycle of receive

640           // store value from interface in register
              if (reg_nfilter){
                *reg_ptr = data_in.read();
              }
              else {
                *reg_fm_ptr = data_in.read();
              }
              sr_count = 0;

        //      //debug
650           value = data_in.read();
              printf("FIR(%u) reading... \t register: %u \t value: %x \t from channel: %u \n",
                  ADDRESS, reg, value, channel);
            }
          }
          return;
        }


        /*******************************
        **          load_imem()          **
660     *******************************/
```

```
void load_imem() {
  // instruction memory file
  if ( (p_ifile = fopen(imem_name, "r")) == NULL)  {
    printf("Error opening %s!!!\n", imem_name);
    abort();
  }
  int i = 0;
  int memword;

  while (fscanf(p_ifile, "%x", &memword) != EOF) {

    imem[i] = memword;
    i++;
  }
  fclose(p_ifile);
  printf("%s loaded\n",imem_name);
}


/*******************************
**          load_cmem()          **
*******************************/
void load_cmem() {
  // coefficient memory file
  if ( (p_cfile = fopen(cmem_name, "r")) == NULL)  {
    printf("Error opening %s!!!\n", cmem_name);
    abort();
  }
  int i = 0;
  int memword;

  while (fscanf(p_cfile, "%x", &memword) != EOF) {

    cmem[i] = memword;
    i++;
  }
  fclose(p_cfile);
  printf("%s loaded\n",cmem_name);
}

/*******************************
**          load_fmem()          **
*******************************/
void load_fmem() {
  // filter memory file
  if ( (p_ffile = fopen(fmem_name, "r")) == NULL)  {
    printf("Error opening %s!!!\n", fmem_name);
    abort();
  }

  int i = 0;
  int j = 0;
  int memword;

  while (fscanf(p_ffile, "%x", &memword) != EOF) {
    fmem[i][j] = memword;
    if (j == 5) {
      j = 0;
      i++;
    }
    else
      j++;
```

```
    }

    fclose(p_cfile);
    printf("%s_loaded\n", fmem_name);
  }

};  // end class
```

A.6 fir_core_node

```
/*****************************************************************************
 **                                                                         **
 ** Name             : fir_core_node.cpp                                   **
 **                                                                         **
 ** Author           : Karsten Larsen, c971833                             **
 ** Date             : 29.12.2002                                          **
 **                                                                         **
 ** Function         : Connects wrapped FIR core with generic OCP node.    **
 **                                                                         **
 ** Major revisions : 08.01.2003   First version.                          **
 **                                                                         **
 *****************************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "ocp_generic_node.cpp"
#include "fir_ocp_wrap.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name, const char *fmem_name, bool STAT_ON, const char *STAT_NAME >
SC_MODULE(fir_core_node) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;          // active low, synchronous reset
  sc_out< sc_uint<DATA_WIDTH> >      data_out;      // data to network
  sc_in< sc_uint<DATA_WIDTH> >       data_in;       // data from network
  sc_out< bool >                     req_out;       // request to network
  sc_in< bool >                      req_in;        // request from network
  sc_out< sc_uint<ADDR_WIDTH> >      addr_out;      // address to network
  sc_in< sc_uint<CHANNEL_WIDTH> >    addr_in;       // address (channel) from network
  sc_in< bool >                      hold_in;       // hold from network
  sc_out< bool >                     hold_out;      // hold to network


  /*********************************
  **          Signals             **
  *********************************/

  sc_signal< sc_uint<3> > nodeM_MCmd;               // OCP signals node master
  sc_signal< sc_uint<CHANNEL_WIDTH> > nodeM_MAddr; // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_MData;      // ...
  sc_signal< bool > nodeM_SCmdAccept;               // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_SData;      // ...
  sc_signal< sc_uint<2> > nodeM_SResp;              // ...
  sc_signal< sc_uint<3> > intM_MCmd;                // OCP signals interface master
```

```
    sc_signal< sc_uint<ADDR_WIDTH> > intM_MAddr;      // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_MData;       // ...
    sc_signal< bool > intM_SCmdAccept;                // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_SData;       // ...
    sc_signal< sc_uint<2> > intM_SResp;               // ...


    /*********************************
    **          Components          **
    *********************************/

    fir_ocp_wrap< ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, fmem_name, STAT_ON
        , STAT_NAME> *fir_wrap;
    ocp_generic_node< DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> *node;


    /*********************************
    **         Constructor          **
    *********************************/

    SC_CTOR(fir_core_node) {


      fir_wrap = new fir_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name,
          fmem_name, STAT_ON, STAT_NAME> ("fir_wrap");
      fir_wrap->clk(clk);
      fir_wrap->nrst(nrst);
      fir_wrap->M_MCmd(intM_MCmd);
      fir_wrap->M_MAddr(intM_MAddr);
      fir_wrap->M_MData(intM_MData);
      fir_wrap->M_SCmdAccept(intM_SCmdAccept);
      fir_wrap->M_SData(intM_SData);
      fir_wrap->M_SResp(intM_SResp);
      fir_wrap->S_MCmd(nodeM_MCmd);
      fir_wrap->S_MAddr(nodeM_MAddr);
      fir_wrap->S_MData(nodeM_MData);
      fir_wrap->S_SCmdAccept(nodeM_SCmdAccept);
      fir_wrap->S_SData(nodeM_SData);
      fir_wrap->S_SResp(nodeM_SResp);


      // Bus nodes

      node = new ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> ("node");
      node->clk(clk);
      node->nrst(nrst);
      node->addr_in(addr_in);
      node->addr_out(addr_out);
      node->data_in(data_in);
      node->data_out(data_out);
      node->hold_out(hold_out);
      node->hold_in(hold_in);
      node->req_out(req_out);
      node->req_in(req_in);
      node->M_MCmd(nodeM_MCmd);
      node->M_MAddr(nodeM_MAddr);
      node->M_MData(nodeM_MData);
      node->M_SCmdAccept(nodeM_SCmdAccept);
      node->M_SData(nodeM_SData);
      node->M_SResp(nodeM_SResp);
      node->S_MCmd(intM_MCmd);
      node->S_MAddr(intM_MAddr);
```

```
    node->S_MData(intM_MData);
    node->S_SCmdAccept(intM_SCmdAccept);
    node->S_SData(intM_SData);
    node->S_SResp(intM_SResp);

  }

}; // END OF fir_core_node
```

A.7 fir_ocp_wrap.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : fir_ocp_wrap.cpp                        **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                 **
 ** Date             : 02.01.2003                              **
 **                                                             **
 ** Function         : Communications wrapper for FIR core. Connects **
 **                  : core with OCP interface.                **
10 **                                                             **
 ** Major revisions : 02.01.2003   First version.              **
 **                  : 14.01.2003   Statistics added.          **
 **                                                             **
 *****************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "fir_core.cpp"
20 #include "ocp_interface.cpp"


template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name, const char *fmem_name, bool STAT_ON, const char *STAT_NAME >
SC_MODULE(fir_ocp_wrap) {


  /***********
   ** Ports **
   ***********/
30

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<3> >              M_MCmd;          // OCP signals as master
  sc_out< sc_uint<ADDR_WIDTH> >     M_MAddr;         // ...
  sc_out< sc_int<DATA_WIDTH> >      M_MData;         // ...
  sc_in< bool >                     M_SCmdAccept;    // ...
  sc_in< sc_int<DATA_WIDTH> >       M_SData;         // ...
  sc_in< sc_uint<2> >               M_SResp;         // ...
  sc_in< sc_uint<3> >               S_MCmd;          // OCP signals as slave
40 sc_in< sc_uint<CHANNEL_WIDTH> >  S_MAddr;         // ...
  sc_in< sc_int<DATA_WIDTH> >       S_MData;         // ...
  sc_out< bool >                    S_SCmdAccept;    // ...
  sc_out< sc_int<DATA_WIDTH> >      S_SData;         // ...
  sc_out< sc_uint<2> >              S_SResp;         // ...


  /*************
   ** Signals **
   *************/
```

```
50
    sc_signal< bool >                    core_rw;          // read/write from core
    sc_signal< sc_uint<DATA_WIDTH> >     core_data_out;    // data from core
    sc_signal< sc_uint<ADDR_WIDTH> >     core_write;       // send address from core
    sc_signal< sc_uint<CHANNEL_WIDTH> >  core_read;        // read address from core
    sc_signal< sc_uint<DATA_WIDTH> >     core_data_in;     // data to core
    sc_signal< bool >                    core_hold;        // hold to core
    sc_signal< bool >                    core_req;         // request from core


60  /****************
     ** Variables  **
     ****************/

    int recv_req_count;
    int send_req_count;
    int hold_count;
    int cc_count;
    FILE *p_stat;

70

    /****************
     ** Components **
     ****************/

    fir_core<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, fmem_name>  *fir;
    ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > *interface;


    SC_CTOR(fir_ocp_wrap) {
80
      // FIR processor
      fir = new fir_core<ADDRESS,DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, fmem_name> (
          "FIR");

      fir->clk(clk);
      fir->nrst(nrst);
      fir->req(core_req);
      fir->rw(core_rw);
      fir->hold(core_hold);
      fir->data_out(core_data_in);
90    fir->write_addr(core_write);
      fir->read_addr(core_read);
      fir->data_in(core_data_out);


      // OCP interface
      interface = new ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > ("
          Interface");

      interface->clk(clk);
      interface->nrst(nrst);
100   interface->core_req(core_req);
      interface->core_rw(core_rw);
      interface->core_data_out(core_data_out);
      interface->core_write(core_write);
      interface->core_read(core_read);
      interface->core_data_in(core_data_in);
      interface->core_hold(core_hold);
      interface->M_MCmd(M_MCmd);
      interface->M_MAddr(M_MAddr);
      interface->M_MData(M_MData);
```

```
110       interface->M_SCmdAccept(M_SCmdAccept);
          interface->M_SData(M_SData);
          interface->M_SResp(M_SResp);
          interface->S_MCmd(S_MCmd);
          interface->S_MAddr(S_MAddr);
          interface->S_MData(S_MData);
          interface->S_SCmdAccept(S_SCmdAccept);
          interface->S_SData(S_SData);
          interface->S_SResp(S_SResp);

120       // statistics generating
          SC_METHOD( statistics );
          sensitive_neg << clk;

        }

        /*******************************
         ** statistics()                **
         ** Outputs hold and req count  **
         ** to file.                    **
130      *******************************/
        void statistics() {

          int i;

          if (nrst == 0) {
            recv_req_count = 0;
            send_req_count = 0;
            hold_count = 0;
            cc_count = 0;
140       }
          else {
            if (STAT_ON) {

              // create statistics
              if ( (p_stat = fopen(STAT_NAME, "w")) == NULL)  {
                printf("Error creating statistics file!!!\n");
                abort();
              }

150           if (core_req) {
                if (core_rw)
                  recv_req_count++;
                else
                  send_req_count++;
              }
              if (core_hold)
                hold_count++;

              fprintf(p_stat,"%u\n%u\n%u\n",recv_req_count, send_req_count, hold_count);
160
              fclose(p_stat);
            }

            cc_count++;

          }
        } // END OF statistics

      };
```

A.8 fir_wrap.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : fir_wrap.cpp                                 **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 17.10.2002                                   **
 **                                                                 **
 ** Function         : Communications wrapper for FIR core. Connects **
 **                  : core with interface.                         **
 **                                                                 **
 ** Major revisions : 17.10.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "fir_core.cpp"
#include "simple_interface.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name, const char *fmem_name >
SC_MODULE(fir_wrap) {

  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > net_send_packet;   // packet to network
  sc_in< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > net_recv_packet; // packet from network
  sc_in< bool >                    send_hold;                   // send hold from network
  sc_out< bool >                   recv_hold;                   // receive hold to
      network
  sc_out< bool >                   send_req;                    // send request to
      network
  sc_in< bool >                    recv_req;                    // receive request from
      network

  /*************
   ** Signals **
   *************/

  sc_signal< bool >                    core_rw;          // read/write from core
  sc_signal< sc_uint<DATA_WIDTH> >     core_data_out;    // data from core
  sc_signal< sc_uint<ADDR_WIDTH> >     core_write;       // send address from core
  sc_signal< sc_uint<CHANNEL_WIDTH> >  core_read;        // read address from core
  sc_signal< sc_uint<DATA_WIDTH> >     core_data_in;     // data to core
  sc_signal< bool >                    core_hold;        // hold to core
  sc_signal< bool >                    core_req;         // request from core

  /****************
   ** Components **
   ****************/

  fir_core<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, fmem_name>  *fir;
  simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > *interface;

  SC_CTOR(fir_wrap) {
```

```
        // FIR processor
        fir = new fir_core<ADDRESS,DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, fmem_name> (
            "FIR");

        fir->clk(clk);
        fir->nrst(nrst);
        fir->req(core_req);
        fir->rw(core_rw);
        fir->hold(core_hold);
        fir->data_out(core_data_in);
        fir->write_addr(core_write);
        fir->read_addr(core_read);
        fir->data_in(core_data_out);

        // Interface
        interface = new simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > ("Interface");

        interface->clk(clk);
        interface->nrst(nrst);
        interface->core_req(core_req);
        interface->core_rw(core_rw);
        interface->core_data_out(core_data_out);
        interface->core_write(core_write);
        interface->core_read(core_read);
        interface->core_data_in(core_data_in);
        interface->core_hold(core_hold);
        interface->net_send_packet(net_send_packet);
        interface->net_recv_packet(net_recv_packet);
        interface->send_hold(send_hold);
        interface->recv_hold(recv_hold);
        interface->send_req(send_req);
        interface->recv_req(recv_req);

    }
};
```

A.9 iir_core.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : iir_core.cpp                                 **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 29.11.2002                                   **
 **                                                                 **
 ** Function         : Behavioral mini-core implementation of a     **
 **                  : dedicated IIR processor.                     **
 ** Notes            : Communicates with external components by send **
 **                  : and receive instructions.                    **
 **                  : Only implements subset of instructions from  **
 **                  : original design.                             **
 **                  : Memories are preloaded from files.           **
 ** Major revisions : 15.11.2002   Complete IIR with communication. **
 **                  : 29.11.2002   4 channels per core added.      **
 **                                                                 **
 *********************************************************************/
```

```
#define SC_INCLUDE_FX

#include "systemc.h"
#include "declarations.h"

template<int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char *
    cmem_name>
SC_MODULE(iir_core) {

  /*********************************
  **          Ports                **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;         // active low, synchronous reset
  sc_out< bool >                     req;          // request to interface
  sc_out< bool >                     rw;           // send or receive (1=rec) to interface
  sc_in< bool >                      hold;         // hold from interface
  sc_out< sc_uint<DATA_WIDTH> >      data_out;     // data to interface
  sc_out< sc_uint<ADDR_WIDTH> >      write_addr;   // send address to interface
  sc_out< sc_uint<CHANNEL_WIDTH> >   read_addr;    // receive address to interface
  sc_in< sc_uint<DATA_WIDTH> >       data_in;      // data from interface


  /*********************************
  **    Signals and Registers     **
  *********************************/

  int PC;                                          // program counter
  sc_uint<5> opcode;
  sc_uint< INSTRUCTION_WIDTH > instruction;
  sc_uint< IMM_WIDTH_IIR > immediate;
  sc_uint< RX_WIDTH > rx;
  sc_uint< RY_WIDTH > ry;
  sc_uint< TARGET_ADDRESS_WIDTH > target_address;
  sc_int< DATA_WIDTH+EXTRA_PRECISION > registers[REGISTER_COUNT];
  sc_fixed< DATA_WIDTH+EXTRA_PRECISION,DATA_INT_WIDTH,SC_RND,SC_SAT >  yN, wN,wNminus1,
      wNminus2;
  sc_fixed< COEFF_LENGTH,COEFF_INT_WIDTH,SC_RND,SC_SAT> a1, a2, b1, b2;
                                  // Coefficients
  sc_fixed< DATA_WIDTH,DATA_INT_WIDTH,SC_RND,SC_SAT >  rx_fx, ry_fx, add_fx;
  bool req_send;       // indicates whether a request has been send in a recieve
      instruction
  int sr_count;        // counter to indicate state of send/receive
  int immediate_sig;
  unsigned int immediate_usig;
  unsigned int reg;
  int value;
  unsigned int send_addr;
  unsigned int recv_addr;
  unsigned int opc;


  /*********************************
  **          Memories             **
  *********************************/

  sc_uint<INSTRUCTION_LENGTH> imem[IMSIZE];
                                                                  // instruction memory
  sc_fixed< DATA_WIDTH+EXTRA_PRECISION,DATA_INT_WIDTH,SC_RND,SC_SAT > bmem[2^BIQUAD_WIDTH
      ][2]; // biquad memory
```

```
sc_fixed< COEFF_LENGTH, COEFF_INT_WIDTH, SC_RND, SC_SAT> cmem[CMSIZE];
                                // coefficient memory

/*********************************
**          Pointers            **
*********************************/

int CP;                                 // coefficient pointer
int w;                                  // pointer into biquad memory


/*********************************
**           Files              **
*********************************/

FILE *p_ifile;
FILE *p_cfile;


/*********************************
**         Constructor          **
*********************************/

SC_CTOR( iir_core ) {

/*********************************
**   Load memories from files   **
*********************************/

  printf("Loading mems from file...\n");

  load_imem();
  load_cmem();

  printf("All files loaded.\n");

  // methods
  SC_METHOD( read_inst );
  sensitive_pos << clk;

}


/*********************************
**          read_inst()         **
**        read instruction      **
*********************************/

void read_inst() {

  if (nrst.read() == 0) {
    req_send = 0;
    CP = 0;
    PC = 0;
    req.write(0);
    sr_count = 0;
  }

  else {  // clockedge
    if (!hold.read()) {           // no hold from interface
      instruction = imem[PC];
      control();                  // decode and perform operation
```

```
      }
      else {
        req.write(1);                // keep req high during hold
      }
    }
    return;
  }


  /*********************************
  **          control()            **
  **  mux instruction and perform **
  **  instruction.                 **
  *********************************/

  void control() {

    int i;

    if (nrst.read() == 0) {          // reset active

    }

    else {                           // no reset

      req.write(0);                  // default value

      opcode = instruction.range(INSTRUCTION_WIDTH-1, INSTRUCTION_WIDTH-5);
      immediate = instruction.range(IMM_WIDTH_IIR-1, 0);
      immediate_usig = immediate;    // ...
      immediate_sig = immediate;     // integer conversions
      opc = opcode;

      if (opc == OP_BIQ_IIR) {
        // type 3 format
        rx = instruction.range(INSTRUCTION_WIDTH-6, INSTRUCTION_WIDTH-RX_WIDTH-5);
        w = instruction.range(INSTRUCTION_WIDTH-RX_WIDTH-6, INSTRUCTION_WIDTH-RX_WIDTH-6-
            BIQUAD_WIDTH);
        ry = instruction.range(INSTRUCTION_WIDTH-RX_WIDTH-BIQUAD_WIDTH-7,
            INSTRUCTION_WIDTH-RX_WIDTH-BIQUAD_WIDTH-RY_WIDTH-6);
      }
      else {
      // type 1 format
      rx = instruction.range(INSTRUCTION_WIDTH-6, INSTRUCTION_WIDTH-RX_WIDTH-5);
      ry = instruction.range(INSTRUCTION_WIDTH-RX_WIDTH-6, INSTRUCTION_WIDTH-RX_WIDTH-
          RY_WIDTH-5);
      reg = rx;  // int conversion for send/receive
      recv_addr = instruction.range(WRITE_WIDTH-1, WRITE_WIDTH-READ_WIDTH);
      send_addr = instruction.range(WRITE_WIDTH-1, 0);

      // type 4 format
      target_address = instruction.range(TARGET_ADDRESS_WIDTH-1, 0);
      }

      req.write(0);  // default

      // decode and execute instruction
      switch(opcode) {  // determine instruction type


      /*********************************
      **          No operation        **
```

```
        ********************************/
        case OP_NOP_IIR:

200       break;


        /********************************
        **              ADD             **
        ********************************/
        case OP_ADD_IIR:

          // fixed point addition
          // this is for test purposes, this is normally not possible
210       for (i=0; i<DATA_WIDTH; i++) {
            rx_fx[i] = (registers[rx])[i];
            ry_fx[i] = (registers[ry])[i];
          }
          add_fx = rx_fx+ry_fx;
          for (i=0; i<DATA_WIDTH; i++) {
              (registers[rx])[i] = add_fx[i];
          }

          // integer addition
220       //      registers[rx] += registers[ry];

          break;


        /********************************
        **          SUBTRACT            **
        ********************************/
        case OP_SUB_IIR:

230       registers[rx] -= registers[ry];

          break;


        /********************************
        **              MOVE            **
        ********************************/
        case OP_MOV_IIR:

240       registers[rx] = registers[ry];

          break;


        /********************************
        **          SHIFT LEFT          **
        ********************************/
        case OP_SHFL_IIR:   // shift left "immediate" times

250       registers[rx].range(DATA_WIDTH+EXTRA_PRECISION-1, immediate_usig) = registers[ry
              ].range(DATA_WIDTH+EXTRA_PRECISION-1-immediate_usig, 0);
          registers[rx].range(immediate_usig-1, 0) = 0;

          break;

        /********************************
        **          SHIFT RIGHT         **
        ********************************/
```

```
        case OP_SHFR_IIR:     // shift right "immediate" times

260       if ( !(registers[ry])[DATA_WIDTH+EXTRA_PRECISION-1] ) { // keep sign bit
            registers[rx].range(DATA_WIDTH+EXTRA_PRECISION-1, DATA_WIDTH+EXTRA_PRECISION-
                immediate_usig) = 0;
          }
          else {
            registers[rx].range(DATA_WIDTH+EXTRA_PRECISION-1, DATA_WIDTH+EXTRA_PRECISION-
                immediate_usig) = 2^immediate;  // insert all 1's
          }
          registers[rx].range(DATA_WIDTH+EXTRA_PRECISION-1-immediate_usig, 0) = registers[
              ry].range(DATA_WIDTH+EXTRA_PRECISION-1, immediate_usig);

          break;

270

        /*********************************
        **              BIQUAD            **
        *********************************/
        case OP_BIQ_IIR:

          if (CP == CMSIZE-1)
            CP = 0;

          // read coefficients
280       a1 = cmem[CP];
          a2 = cmem[CP+1];
          b1 = cmem[CP+2];
          b2 = cmem[CP+3];

          // read data
          wNminus1 = bmem[w][0];
          wNminus2 = bmem[w][1];

          // biquad equation 1
290       wN = a1*wNminus1 + a2*wNminus2 + registers[ry];

          // biquad equation 2
          yN = wN + b1*wNminus1 + b2*wNminus2;

          // store in register while converting to bitvector
          for (i=0; i<DATA_WIDTH+EXTRA_PRECISION; i++) {
              (registers[rx])[i] = yN[i];
          }

300       // shift biquad delay line
          bmem[w][1] = bmem[w][0];
          bmem[w][0] = wN;

          break;


        /*********************************
        **              JUMP              **
        *********************************/
310     case OP_JUMP_IIR:

          PC = target_address;
          PC--;

          break;
```

```
        /*********************************
        **              send             **
        *********************************/
        case OP_SEND_IIR:

          send(reg,send_addr);

          break;


        /*********************************
        **             receive           **
        *********************************/
        case OP_RECEIVE_IIR:

          receive(reg,recv_addr);

          break;


        default:

          printf("No such IIR(%u) operation!\n", ADDRESS);
          break;
        }

        PC++;
      }

      return;
    }


    /*********************************
     **         send(reg,imm)         **
     ** Sends contents of reg to    **
     ** output channel indicated by **
     ** imm.                        **
     *********************************/

    void send(int reg, int channel) {

      if (sr_count == 0) {

        value = registers[reg];

        // pass send request to interface
        write_addr.write(channel);
        data_out.write(value);
        req.write(1);
        rw.write(0);

        PC--;
        sr_count++;

        //debugging
        printf("IIR(%u) sending... \t register: %u \t value: %x \t to channel: %u \n",
            ADDRESS, reg, value, channel);
      }
      else {
        if (sr_count == 1) {    //2nd clockcycle
```

```
        PC--;
        req.write(0);
        sr_count++;
      }
      else {                    // 3rd clockcycle
        sr_count = 0;
      }
    }
    return;
  }


/*********************************
 **       receive(reg,imm)       **
 ** Recieves data package from   **
 ** channel indicated by imm     **
 ** and stores it in register    **
 ** indicated by reg.            **
 *********************************/

void receive(int reg, int channel) {

  if (sr_count == 0) { // 1st cycle of receive
    // send request to interface
    req.write(1);
    rw.write(1);
    read_addr.write(channel);
    // make sure 2nd cycle is performed
    PC--;
    sr_count++;
  }

  else {        // 2nd cycle
    if (sr_count == 1) {
      PC--;
      req.write(0);
      sr_count++;
    }

    else {      // 3rd and last cycle of receive

      // store value from interface in register
      registers[reg] = data_in.read();

      sr_count = 0;

      //debug
      value = data_in.read();
      printf("IIR(%u) reading... \t register: %u \t value: %x \t from channel: %u \n",
          ADDRESS, reg, value, channel);
    }
  }
  return;
}


/*********************************
**          load_imem()          **
*********************************/
void load_imem() {
  // instruction memory file
```

```
    if ( (p_ifile = fopen(imem_name, "r")) == NULL)  {
      printf("Error opening %s!!!\n", imem_name);
      abort();
    }
    int i = 0;
    int intword;

    while (fscanf(p_ifile, "%x", &intword) != EOF) {
      imem[i] = intword;
      i++;
    }
    fclose(p_ifile);
    printf("%s loaded\n",imem_name);
  }


  /*********************************
  **          load_cmem()          **
  *********************************/
  void load_cmem() {
    // coefficient memory file
    if ( (p_cfile = fopen(cmem_name, "r")) == NULL)  {
      printf("Error opening %s!!!\n", cmem_name);
      abort();
    }
    int i = 0;
    //    double dblword;
    int dblword;

    while (fscanf(p_cfile, "%x", &dblword) != EOF) {
      cmem[i] = dblword;
      i++;
      //       printf("%u \t %x\n",i,dblword);
    }
    fclose(p_cfile);
    printf("%s loaded\n",cmem_name);
  }

};  // end class
```

A.10 iir_core_node

```
/*****************************************************************************
 **                                                                         **
 ** Name             : iir_core_node.cpp                                   **
 **                                                                         **
 ** Author           : Karsten Larsen, c971833                             **
 ** Date             : 29.12.2002                                          **
 **                                                                         **
 ** Function         : Connects wrapped IIR core with generic OCP node.    **
 **                                                                         **
 ** Major revisions : 08.01.2003   Final version.                          **
 **                                                                         **
 *****************************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "ocp_generic_node.cpp"
#include "iir_ocp_wrap.cpp"
```

```
20
template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name, bool STAT_ON, const char *STAT_NAME >
SC_MODULE(iir_core_node) {

  /*********************************
  **          Ports                **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;            // active low, synchronous reset
30 sc_out< sc_uint<DATA_WIDTH> >      data_out;        // data to network
  sc_in< sc_uint<DATA_WIDTH> >       data_in;         // data from network
  sc_out< bool >                     req_out;         // request to network
  sc_in< bool >                      req_in;          // request from network
  sc_out< sc_uint<ADDR_WIDTH> >      addr_out;        // address to network
  sc_in< sc_uint<CHANNEL_WIDTH> >    addr_in;         // address (channel) from network
  sc_in< bool >                      hold_in;         // hold from network
  sc_out< bool >                     hold_out;        // hold to network


40 /*********************************
  **          Signals              **
  *********************************/

  sc_signal< sc_uint<3> > nodeM_MCmd;              // OCP signals node master
  sc_signal< sc_uint<CHANNEL_WIDTH> > nodeM_MAddr; // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_MData;     // ...
  sc_signal< bool > nodeM_SCmdAccept;              // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_SData;     // ...
  sc_signal< sc_uint<2> > nodeM_SResp;             // ...
50 sc_signal< sc_uint<3> > intM_MCmd;               // OCP signals interface master
  sc_signal< sc_uint<ADDR_WIDTH> > intM_MAddr;     // ...
  sc_signal< sc_int<DATA_WIDTH> > intM_MData;      // ...
  sc_signal< bool > intM_SCmdAccept;               // ...
  sc_signal< sc_int<DATA_WIDTH> > intM_SData;      // ...
  sc_signal< sc_uint<2> > intM_SResp;              // ...


  /*********************************
  **          Components           **
60 *********************************/

  iir_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name, STAT_ON, STAT_NAME
      > *iir_wrap;
  ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> *node;


  /*********************************
  **         Constructor           **
  *********************************/

70 SC_CTOR(iir_core_node) {

    iir_wrap = new iir_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name,
        STAT_ON, STAT_NAME> ("wrap");
    iir_wrap->clk(clk);
    iir_wrap->nrst(nrst);
    iir_wrap->M_MCmd(intM_MCmd);
    iir_wrap->M_MAddr(intM_MAddr);
    iir_wrap->M_MData(intM_MData);
    iir_wrap->M_SCmdAccept(intM_SCmdAccept);
```

```
    iir_wrap->M_SData(intM_SData);
    iir_wrap->M_SResp(intM_SResp);
    iir_wrap->S_MCmd(nodeM_MCmd);
    iir_wrap->S_MAddr(nodeM_MAddr);
    iir_wrap->S_MData(nodeM_MData);
    iir_wrap->S_SCmdAccept(nodeM_SCmdAccept);
    iir_wrap->S_SData(nodeM_SData);
    iir_wrap->S_SResp(nodeM_SResp);


    // Bus nodes

    node = new ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> ("node");
    node->clk(clk);
    node->nrst(nrst);
    node->addr_in(addr_in);
    node->addr_out(addr_out);
    node->data_in(data_in);
    node->data_out(data_out);
    node->hold_out(hold_out);
    node->hold_in(hold_in);
    node->req_out(req_out);
    node->req_in(req_in);
    node->M_MCmd(nodeM_MCmd);
    node->M_MAddr(nodeM_MAddr);
    node->M_MData(nodeM_MData);
    node->M_SCmdAccept(nodeM_SCmdAccept);
    node->M_SData(nodeM_SData);
    node->M_SResp(nodeM_SResp);
    node->S_MCmd(intM_MCmd);
    node->S_MAddr(intM_MAddr);
    node->S_MData(intM_MData);
    node->S_SCmdAccept(intM_SCmdAccept);
    node->S_SData(intM_SData);
    node->S_SResp(intM_SResp);

  }
}; // END OF iir_core_node
```

A.11 iir_ocp_wrap.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : iir_ocp_wrap.cpp                         **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                  **
 ** Date             : 02.01.2003                               **
 **                                                             **
 ** Function         : Communications wrapper for IIR core. Connects **
 **                  : core with OCP interface.                 **
 **                                                             **
 ** Major revisions : 02.01.2003    First version.             **
 **                  : 14.01.2003    Statistics added.          **
 **                                                             **
 *****************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "iir_core.cpp"
#include "ocp_interface.cpp"
```

```
template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name, bool STAT_ON, const char *STAT_NAME >
SC_MODULE(iir_ocp_wrap) {


  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<3> >              M_MCmd;          // OCP signals as master
  sc_out< sc_uint<ADDR_WIDTH> >     M_MAddr;         // ...
  sc_out< sc_int<DATA_WIDTH> >      M_MData;         // ...
  sc_in< bool >                     M_SCmdAccept;    // ...
  sc_in< sc_int<DATA_WIDTH> >       M_SData;         // ...
  sc_in< sc_uint<2> >               M_SResp;         // ...
  sc_in< sc_uint<3> >               S_MCmd;          // OCP signals as slave
  sc_in< sc_uint<CHANNEL_WIDTH> >   S_MAddr;         // ...
  sc_in< sc_int<DATA_WIDTH> >       S_MData;         // ...
  sc_out< bool >                    S_SCmdAccept;    // ...
  sc_out< sc_int<DATA_WIDTH> >      S_SData;         // ...
  sc_out< sc_uint<2> >              S_SResp;         // ...


  /*************
   ** Signals **
   *************/

  sc_signal< bool >                     core_rw;          // read/write from core
  sc_signal< sc_uint<DATA_WIDTH> >      core_data_out;    // data from core
  sc_signal< sc_uint<ADDR_WIDTH> >      core_write;       // send address from core
  sc_signal< sc_uint<CHANNEL_WIDTH> >   core_read;        // read address from core
  sc_signal< sc_uint<DATA_WIDTH> >      core_data_in;     // data to core
  sc_signal< bool >                     core_hold;        // hold to core
  sc_signal< bool >                     core_req;         // request from core


  /****************
   ** Variables  **
   ****************/

  int recv_req_count;
  int send_req_count;
  int hold_count;
  int cc_count;
  FILE *p_stat;


  /****************
   ** Components **
   ****************/

  iir_core<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name>  *iir;
  ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > *interface;


  SC_CTOR(iir_ocp_wrap) {

    // IIR processor
```

```
    iir = new iir_core<ADDRESS,DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name> ("IIR");

    iir->clk(clk);
    iir->nrst(nrst);
    iir->req(core_req);
    iir->rw(core_rw);
    iir->hold(core_hold);
    iir->data_out(core_data_in);
    iir->write_addr(core_write);
    iir->read_addr(core_read);
    iir->data_in(core_data_out);


    // OCP interface
    interface = new ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > ("
        Interface");

    interface->clk(clk);
    interface->nrst(nrst);
    interface->core_req(core_req);
    interface->core_rw(core_rw);
    interface->core_data_out(core_data_out);
    interface->core_write(core_write);
    interface->core_read(core_read);
    interface->core_data_in(core_data_in);
    interface->core_hold(core_hold);
    interface->M_MCmd(M_MCmd);
    interface->M_MAddr(M_MAddr);
    interface->M_MData(M_MData);
    interface->M_SCmdAccept(M_SCmdAccept);
    interface->M_SData(M_SData);
    interface->M_SResp(M_SResp);
    interface->S_MCmd(S_MCmd);
    interface->S_MAddr(S_MAddr);
    interface->S_MData(S_MData);
    interface->S_SCmdAccept(S_SCmdAccept);
    interface->S_SData(S_SData);
    interface->S_SResp(S_SResp);

    // statistics generating
    SC_METHOD( statistics ); // Creates run-time report and statistics
    sensitive_neg << clk;

}

 /*********************************
  ** statistics()                 **
  ** Outputs hold and req count   **
  **  to file.                    **
  *********************************/
void statistics() {

  int i;

  if (nrst == 0) {
    recv_req_count = 0;
    send_req_count = 0;
    hold_count = 0;
    cc_count = 0;
  }
  else {
    if (STAT_ON) {
```

```
        // create statistics
        if ( (p_stat = fopen(STAT_NAME, "w")) == NULL)  {
          printf("Error creating statistics file!!!\n");
          abort();
        }

        if (core_req) {
          if (core_rw)
            recv_req_count++;
          else
            send_req_count++;
        }
        if (core_hold)
          hold_count++;

        fprintf(p_stat,"%u\n%u\n%u\n", recv_req_count, send_req_count, hold_count);

        fclose(p_stat);
      }

      cc_count++;

    }
  } // END OF statistics

};
```

A.12 iir_wrap.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : iir_wrap.cpp                                 **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 17.11.2002                                   **
 **                                                                 **
 ** Function         : Communications wrapper for IIR core. Connects **
 **                  : corewith interface.                          **
 **                                                                 **
 ** Major revisions : 17.11.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "iir_core.cpp"
#include "simple_interface.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *imem_name, const char
    *cmem_name >
SC_MODULE(iir_wrap) {

  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
```

```
    sc_out< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > net_send_packet;   // packet to network
    sc_in< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > net_recv_packet; // packet from network
    sc_in< bool >                   send_hold;                    // send hold from network
    sc_out< bool >                  recv_hold;                    // receive hold to
        network
    sc_out< bool >                  send_req;                     // send request to
        network
    sc_in< bool >                   recv_req;                     // receive request from
        network

    /*************
     ** Signals **
40   *************/

    sc_signal< bool >                    core_rw;          // read/write from core
    sc_signal< sc_uint<DATA_WIDTH> >     core_data_out;    // data from core
    sc_signal< sc_uint<ADDR_WIDTH> >     core_write;       // send address from core
    sc_signal< sc_uint<CHANNEL_WIDTH> >  core_read;        // receive address from core
    sc_signal< sc_uint<DATA_WIDTH> >     core_data_in;     // data to core
    sc_signal< bool >                    core_hold;        // hold to core
    sc_signal< bool >                    core_req;         // request from core

50  /****************
     ** Components **
     ****************/

    iir_core<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name>  *iir;
    simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > *interface;

    SC_CTOR(iir_wrap) {

      // IIR processor
60    iir = new iir_core<ADDRESS, DATA_WIDTH, ADDR_WIDTH, imem_name, cmem_name> ("IIR");

      iir->clk(clk);
      iir->nrst(nrst);
      iir->req(core_req);
      iir->rw(core_rw);
      iir->hold(core_hold);
      iir->data_out(core_data_in);
      iir->write_addr(core_write);
      iir->data_in(core_data_out);
70    iir->read_addr(core_read);


      // Interface
      interface = new simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > ("Interface");

      interface->clk(clk);
      interface->nrst(nrst);
      interface->core_req(core_req);
      interface->core_rw(core_rw);
80    interface->core_data_out(core_data_out);
      interface->core_write(core_write);
      interface->core_read(core_read);
      interface->core_data_in(core_data_in);
      interface->core_hold(core_hold);
      interface->net_send_packet(net_send_packet);
      interface->net_recv_packet(net_recv_packet);
      interface->send_hold(send_hold);
      interface->recv_hold(recv_hold);
      interface->send_req(send_req);
```

```
90      interface->recv_req(recv_req);

    }
};
```

A.13 input_core.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : input_core.cpp                               **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 12.12.2002                                   **
 **                                                                 **
 ** Function         : Mini-core simulating external data transmitter. **
 **                  : Outputs data from file.                      **
10 ** Major revisions : 12.12.2002   Final version.                 **
 **                                                                 **
 *********************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "declarations.h"

20 #define INPUT_LENGTH 300
#define CC 50

template<int CHANNEL, int DATA_WIDTH, int ADDR_WIDTH, const char *input_name>
SC_MODULE(input_core) {

  /*********************************
  **          Ports               **
  *********************************/

30  sc_in_clk                          clk;
    sc_in< bool >                      nrst;        // active low, synchronous reset
    sc_out< bool >                     req;         // request to interface
    sc_out< bool >                     rw;          // send or receive (1=rec) to interface
    sc_in< bool >                      hold;        // hold from interface
    sc_out< sc_uint<DATA_WIDTH> >      data_out;    // data to interface
    sc_out< sc_uint<ADDR_WIDTH> >      write_addr;  // send address to interface
    sc_out< sc_uint<CHANNEL_WIDTH> >   read_addr;   // read channel to interface
    sc_in< sc_uint<DATA_WIDTH> >       data_in;     // data from interface

40  FILE *p_infile;
    sc_uint<DATA_WIDTH> inputs[INPUT_LENGTH];
    sc_fixed< DATA_WIDTH,SAMPLE_INT_WIDTH,SC_RND,SC_SAT > datain_fx;
    sc_uint<DATA_WIDTH> input_temp;
    int counter;
    int index;
    int sr_count;
    int DATA_COUNT;
    int count;

50  SC_CTOR( input_core ) {

      int i,j;
      double memword;
```

```
    for (i=0; i<INPUT_LENGTH; i++)
      inputs[i] = 0;

    // open input file
    if ( (p_infile = fopen(input_name, "r")) == NULL)  {
      printf("Error opening %s!!!\n", input_name);
      abort();
    }
    i = 0;
    DATA_COUNT = 0;
    while (fscanf(p_infile, "%lf", &memword) != EOF) {
      datain_fx = memword;                              // convert to fixed point
      for (j=0; j<DATA_WIDTH; j++) {                    // convert to bit vector
        (inputs[i])[j] = datain_fx[j];
      }
      i++;
      DATA_COUNT++;
    }
    fclose(p_infile);
    printf("%s loaded\n",input_name);

    SC_METHOD( read );
    sensitive_pos << clk;

  } // constructor

  void read() {

     if (nrst.read() == 0) {
      req.write(0);
      counter = 0;
      index = 0;
      sr_count = 0;
      count = 0;
    }

    else {  // clockedge
      req.write(0);
      if (index < DATA_COUNT) {  // still data left
        if (!hold.read()) {      // no hold from interface
          if (counter == CC) {
            send(inputs[index]);
            index++;
            counter = 0;
          }
          else {
            counter++;
          }
        }
      }
      else
        req.write(0);
    }
    return;
  }

  void send(sc_uint<DATA_WIDTH> value) {

      // pass send request to interface
      write_addr.write(CHANNEL);
      data_out.write(value);
```

```
        req.write(1);
        rw.write(0);

        //debugging
        int val_int = value;
        printf("INPUT sending... \t value: %x \t to channel: %u \n", val_int, CHANNEL);

      return;
    }

}; // class
```

A.14 input_core_node

```
/*****************************************************************
 **                                                             **
 ** Name             : input_core_node.cpp                      **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                  **
 ** Date             : 29.12.2002                               **
 **                                                             **
 ** Function         : Connects wrapped input core with generic OCP  **
 **                  : node.                                    **
 **                                                             **
 ** Major revisions : 08.01.2003   First version.              **
 **                                                             **
 *****************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "ocp_generic_node.cpp"
#include "input_ocp_wrap.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *input_name, bool
    STAT_ON, const char *stat_name >
SC_MODULE(input_core_node) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;          // active low, synchronous reset
  sc_out< sc_uint<DATA_WIDTH> >      data_out;      // data to network
  sc_in< sc_uint<DATA_WIDTH> >       data_in;       // data from network
  sc_out< bool >                     req_out;       // request to network
  sc_in< bool >                      req_in;        // request from network
  sc_out< sc_uint<ADDR_WIDTH> >      addr_out;      // address to network
  sc_in< sc_uint<CHANNEL_WIDTH> >    addr_in;       // address (channel) from network
  sc_in< bool >                      hold_in;       // hold from network
  sc_out< bool >                     hold_out;      // hold to network


  /*********************************
  **          Signals             **
  *********************************/

  sc_signal< sc_uint<3> > nodeM_MCmd;               // OCP signals node master
  sc_signal< sc_uint<CHANNEL_WIDTH> > nodeM_MAddr; // ...
```

```
    sc_signal< sc_int<DATA_WIDTH> > nodeM_MData;      // ...
    sc_signal< bool > nodeM_SCmdAccept;               // ...
    sc_signal< sc_int<DATA_WIDTH> > nodeM_SData;      // ...
    sc_signal< sc_uint<2> > nodeM_SResp;              // ...
    sc_signal< sc_uint<3> > intM_MCmd;                // OCP signals interface master
    sc_signal< sc_uint<ADDR_WIDTH> > intM_MAddr;      // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_MData;       // ...
    sc_signal< bool > intM_SCmdAccept;                // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_SData;       // ...
    sc_signal< sc_uint<2> > intM_SResp;               // ...


    /*********************************
    **          Components          **
    *********************************/

    input_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, input_name, STAT_ON, stat_name> *
        input_wrap;
    ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> *node;


    /*********************************
    **         Constructor          **
    *********************************/

    SC_CTOR(input_core_node) {


      input_wrap = new input_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, input_name, STAT_ON
          , stat_name> ("input_wrap");
      input_wrap->clk(clk);
      input_wrap->nrst(nrst);
      input_wrap->M_MCmd(intM_MCmd);
      input_wrap->M_MAddr(intM_MAddr);
      input_wrap->M_MData(intM_MData);
      input_wrap->M_SCmdAccept(intM_SCmdAccept);
      input_wrap->M_SData(intM_SData);
      input_wrap->M_SResp(intM_SResp);
      input_wrap->S_MCmd(nodeM_MCmd);
      input_wrap->S_MAddr(nodeM_MAddr);
      input_wrap->S_MData(nodeM_MData);
      input_wrap->S_SCmdAccept(nodeM_SCmdAccept);
      input_wrap->S_SData(nodeM_SData);
      input_wrap->S_SResp(nodeM_SResp);


      // Bus nodes

      node = new ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> ("node");
      node->clk(clk);
      node->nrst(nrst);
      node->addr_in(addr_in);
      node->addr_out(addr_out);
      node->data_in(data_in);
      node->data_out(data_out);
      node->hold_out(hold_out);
      node->hold_in(hold_in);
      node->req_out(req_out);
      node->req_in(req_in);
      node->M_MCmd(nodeM_MCmd);
      node->M_MAddr(nodeM_MAddr);
      node->M_MData(nodeM_MData);
```

```
    node->M_SCmdAccept(nodeM_SCmdAccept);
    node->M_SData(nodeM_SData);
    node->M_SResp(nodeM_SResp);
    node->S_MCmd(intM_MCmd);
    node->S_MAddr(intM_MAddr);
    node->S_MData(intM_MData);
    node->S_SCmdAccept(intM_SCmdAccept);
    node->S_SData(intM_SData);
    node->S_SResp(intM_SResp);

  }
}; // END OF input_core_node
```

A.15 input_ocp_wrap.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : input_ocp_wrap.cpp                       **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                  **
 ** Date             : 02.01.2003                               **
 **                                                             **
 ** Function         : Communications wrapper for input core. Connects **
 **                  :  corewith OCP interface.                 **
 **                                                             **
 ** Major revisions : 02.01.2003   First version.              **
 **                  : 14.01.2003   Statistics added.           **
 **                                                             **
 *****************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "input_core.cpp"
#include "ocp_interface.cpp"


template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *input_name, bool
    STAT_ON, const char *STAT_NAME >
SC_MODULE(input_ocp_wrap) {


  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<3> >              M_MCmd;          // OCP signals as master
  sc_out< sc_uint<ADDR_WIDTH> >     M_MAddr;         // ...
  sc_out< sc_int<DATA_WIDTH> >      M_MData;         // ...
  sc_in< bool >                     M_SCmdAccept;    // ...
  sc_in< sc_int<DATA_WIDTH> >       M_SData;         // ...
  sc_in< sc_uint<2> >               M_SResp;         // ...
  sc_in< sc_uint<3> >               S_MCmd;          // OCP signals as slave
  sc_in< sc_uint<CHANNEL_WIDTH> >   S_MAddr;         // ...
  sc_in< sc_int<DATA_WIDTH> >       S_MData;         // ...
  sc_out< bool >                    S_SCmdAccept;    // ...
  sc_out< sc_int<DATA_WIDTH> >      S_SData;         // ...
  sc_out< sc_uint<2> >              S_SResp;         // ...
```

```
/*************
 ** Signals **
 *************/

sc_signal< bool >                    core_rw;          // read/write from core
sc_signal< sc_uint<DATA_WIDTH> >     core_data_out;    // data from core
sc_signal< sc_uint<ADDR_WIDTH> >     core_write;       // send address from core
sc_signal< sc_uint<CHANNEL_WIDTH> >  core_read;        // read address from core
sc_signal< sc_uint<DATA_WIDTH> >     core_data_in;     // data to core
sc_signal< bool >                    core_hold;        // hold to core
sc_signal< bool >                    core_req;         // request from core


/***************
 ** Variables  **
 ***************/

int recv_req_count;
int send_req_count;
int hold_count;
int cc_count;
FILE *p_stat;


/***************
 ** Components **
 ***************/

input_core<0, DATA_WIDTH, ADDR_WIDTH, input_name > *input;
ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > *interface;


SC_CTOR(input_ocp_wrap) {

  // Input core
  input = new input_core< 0, DATA_WIDTH, ADDR_WIDTH, input_name > ("input");

  input->clk(clk);
  input->nrst(nrst);
  input->req(core_req);
  input->rw(core_rw);
  input->hold(core_hold);
  input->data_out(core_data_in);
  input->write_addr(core_write);
  input->read_addr(core_read);
  input->data_in(core_data_out);


  // Interface
  interface = new ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > ("
      Interface");

  interface->clk(clk);
  interface->nrst(nrst);
  interface->core_req(core_req);
  interface->core_rw(core_rw);
  interface->core_data_out(core_data_out);
  interface->core_write(core_write);
  interface->core_read(core_read);
  interface->core_data_in(core_data_in);
  interface->core_hold(core_hold);
```

```
    interface->M_MCmd(M_MCmd);
    interface->M_MAddr(M_MAddr);
    interface->M_MData(M_MData);
    interface->M_SCmdAccept(M_SCmdAccept);
    interface->M_SData(M_SData);
    interface->M_SResp(M_SResp);
    interface->S_MCmd(S_MCmd);
    interface->S_MAddr(S_MAddr);
    interface->S_MData(S_MData);
    interface->S_SCmdAccept(S_SCmdAccept);
    interface->S_SData(S_SData);
    interface->S_SResp(S_SResp);

    // statistics generating
    SC_METHOD( statistics ); // Creates run-time report and statistics
    sensitive_neg << clk;

  }

  /*********************************
   ** statistics()                 **
   ** Outputs hold and req count   **
   **  to file.                    **
   *********************************/
  void statistics() {

    int i;

    if (nrst == 0) {
      recv_req_count = 0;
      send_req_count = 0;
      hold_count = 0;
      cc_count = 0;
    }
    else {
      if (STAT_ON) {

        // create statistics
        if ( (p_stat = fopen(STAT_NAME, "w")) == NULL)  {
          printf("Error creating statistics file!!!\n");
          abort();
        }

        if (core_req) {
          if (core_rw)
            recv_req_count++;
          else
            send_req_count++;
        }
        if (core_hold)
          hold_count++;

        fprintf(p_stat,"%u\n%u\n%u\n", recv_req_count, send_req_count, hold_count);

        fclose(p_stat);
      }

      cc_count++;

    }
  } // END OF statistics
```

```
};
```

A.16 input_wrap.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name            : input_wrap.cpp                                **
 **                                                                 **
 ** Author          : Karsten Larsen, c971833                       **
 ** Date            : 13.12.2002                                    **
 **                                                                 **
 ** Function        : Communications wrapper for input core. Connects **
 **                 : core with interface.                          **
 **                                                                 **
 ** Major revisions : 13.12.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "input_core.cpp"
#include "simple_interface.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *input_name >
SC_MODULE(input_wrap) {

  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > net_send_packet;   // packet to network
  sc_in< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > net_recv_packet; // packet from network
  sc_in< bool >                    send_hold;                   // send hold from network
  sc_out< bool >                   recv_hold;                   // receive hold to
      network
  sc_out< bool >                   send_req;                    // send request to
      network
  sc_in< bool >                    recv_req;                    // receive request from
      network

  /*************
   ** Signals **
   *************/

  sc_signal< bool >                     core_rw;          // read/write from core
  sc_signal< sc_uint<DATA_WIDTH> >      core_data_out;    // data from core
  sc_signal< sc_uint<ADDR_WIDTH> >      core_write;       // send address from core
  sc_signal< sc_uint<CHANNEL_WIDTH> >   core_read;        // read address from core
  sc_signal< sc_uint<DATA_WIDTH> >      core_data_in;     // data to core
  sc_signal< bool >                     core_hold;        // hold to core
  sc_signal< bool >                     core_req;         // request from core

  /***************
   ** Components **
   ***************/

  input_core<0, DATA_WIDTH, ADDR_WIDTH, input_name > *input;
  simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > *interface;
```

```
  SC_CTOR(input_wrap) {

    // Input core
    input = new input_core< 0, DATA_WIDTH, ADDR_WIDTH, input_name > ("input");

    input->clk(clk);
    input->nrst(nrst);
    input->req(core_req);
    input->rw(core_rw);
    input->hold(core_hold);
    input->data_out(core_data_in);
    input->write_addr(core_write);
    input->read_addr(core_read);
    input->data_in(core_data_out);

    // Interface
    interface = new simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > ("Interface");

    interface->clk(clk);
    interface->nrst(nrst);
    interface->core_req(core_req);
    interface->core_rw(core_rw);
    interface->core_data_out(core_data_out);
    interface->core_write(core_write);
    interface->core_read(core_read);
    interface->core_data_in(core_data_in);
    interface->core_hold(core_hold);
    interface->net_send_packet(net_send_packet);
    interface->net_recv_packet(net_recv_packet);
    interface->send_hold(send_hold);
    interface->recv_hold(recv_hold);
    interface->send_req(send_req);
    interface->recv_req(recv_req);

  }
};
```

A.17 main.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : main.cpp                                **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                 **
 ** Date             : 15.01.2002                              **
 **                                                             **
 ** Function         : Main module for mini-core system. Supplies **
 **                  : clock and  reset for network (reset from **
 **                  : stimuli.cpp).                           **
 ** Notes            : Traces important signals, written to VCD file. **
 **                                                             **
 ** Major revisions : 15.01.2002   Final version.             **
 **                                                             **
 *****************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
```

```
#include "declarations.h"

#include "stimuli.cpp"
#include "core_system.cpp"

// filenames

// Input and output
char input_name[] = "input.bin";
char output_name[] = "output.bin";
// FIRs
char imem_name1[] = "imem1.bin";
char cmem_name1[] = "cmem1.bin";
char fmem_name1[] = "fmem1.bin";
char imem_name2[] = "imem2.bin";
char cmem_name2[] = "cmem2.bin";
char fmem_name2[] = "fmem2.bin";
char imem_name3[] = "imem3.bin";
char cmem_name3[] = "cmem3.bin";
char fmem_name3[] = "fmem3.bin";
//IIRs
char imem_name4[] = "imem4.bin";
char cmem_name4[] = "cmem4.bin";
char imem_name5[] = "imem5.bin";
char cmem_name5[] = "cmem5.bin";
char imem_name6[] = "imem6.bin";
char cmem_name6[] = "cmem6.bin";

// Statistics files
char input_stat_name[] = "stat_input.txt";
char output_stat_name[] = "stat_output.txt";
char fir1_stat_name[] = "stat_fir1.txt";
char fir2_stat_name[] = "stat_fir2.txt";
char fir3_stat_name[] = "stat_fir3.txt";
char iir1_stat_name[] = "stat_iir1.txt";
char iir2_stat_name[] = "stat_iir2.txt";
char iir3_stat_name[] = "stat_iir3.txt";

// components
core_system<3, 3, 16, 16, imem_name1, cmem_name1, fmem_name1, imem_name2, cmem_name2,
    fmem_name2, imem_name3, cmem_name3, fmem_name3, imem_name4, cmem_name4, imem_name5,
    cmem_name5, imem_name6, cmem_name6, input_name, output_name, 1, input_stat_name,
    output_stat_name, fir1_stat_name, fir2_stat_name, fir3_stat_name, iir1_stat_name,
    iir2_stat_name, iir3_stat_name> *sys;
stimuli *nrst_gen;

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

  // signals
  sc_signal< bool > nrst;

  //clock
  sc_clock clk("clk", 20);

  //connnections
  sys = new core_system<3, 3, 16, 16, imem_name1, cmem_name1, fmem_name1, imem_name2,
      cmem_name2, fmem_name2, imem_name3, cmem_name3, fmem_name3, imem_name4, cmem_name4
      , imem_name5, cmem_name5, imem_name6, cmem_name6, input_name, output_name, 1,
      input_stat_name, output_stat_name, fir1_stat_name, fir2_stat_name, fir3_stat_name,
      iir1_stat_name, iir2_stat_name, iir3_stat_name> ("sys");
  sys->nrst(nrst);
  sys->clk(clk);
```

```
    nrst_gen = new stimuli ("nrst_gen");
    nrst_gen->nrst(nrst);
    nrst_gen->clk(clk);

    //tracing
    sc_trace_file *tf = sc_create_vcd_trace_file("trace_fir");

    sc_trace(tf, clk,"clk");
    sc_trace(tf, nrst, "nrst");

    sc_trace(tf, sys->bus->req_out[0], "req_out0");
    sc_trace(tf, sys->bus->req_out[1], "req_out1");
    sc_trace(tf, sys->bus->req_out[2], "req_out2");
    sc_trace(tf, sys->bus->req_out[3], "req_out3");
    sc_trace(tf, sys->bus->req_out[4], "req_out4");
    sc_trace(tf, sys->bus->req_out[5], "req_out5");
    sc_trace(tf, sys->bus->req_out[6], "req_out6");
    sc_trace(tf, sys->bus->req_out[7], "req_out7");
    sc_trace(tf, sys->bus->req_in[0], "req_in0");
    sc_trace(tf, sys->bus->req_in[1], "req_in1");
    sc_trace(tf, sys->bus->req_in[2], "req_in2");
    sc_trace(tf, sys->bus->req_in[3], "req_in3");
    sc_trace(tf, sys->bus->req_in[4], "req_in4");
    sc_trace(tf, sys->bus->req_in[5], "req_in5");
    sc_trace(tf, sys->bus->req_in[6], "req_in6");
    sc_trace(tf, sys->bus->req_in[7], "req_in7");
    sc_trace(tf, sys->bus->hold_out[0], "hold_out0");
    sc_trace(tf, sys->bus->hold_out[1], "hold_out1");
    sc_trace(tf, sys->bus->hold_out[2], "hold_out2");
    sc_trace(tf, sys->bus->hold_out[3], "hold_out3");
    sc_trace(tf, sys->bus->hold_out[4], "hold_out4");
    sc_trace(tf, sys->bus->hold_out[5], "hold_out5");
    sc_trace(tf, sys->bus->hold_out[6], "hold_out6");
    sc_trace(tf, sys->bus->hold_out[7], "hold_out7");
    sc_trace(tf, sys->bus->hold_in[0], "hold_in0");
    sc_trace(tf, sys->bus->hold_in[1], "hold_in1");
    sc_trace(tf, sys->bus->hold_in[2], "hold_in2");
    sc_trace(tf, sys->bus->hold_in[3], "hold_in3");
    sc_trace(tf, sys->bus->hold_in[4], "hold_in4");
    sc_trace(tf, sys->bus->hold_in[5], "hold_in5");
    sc_trace(tf, sys->bus->hold_in[6], "hold_in6");
    sc_trace(tf, sys->bus->hold_in[7], "hold_in7");
    sc_trace(tf, sys->bus->data_in[0], "data_in0");
    sc_trace(tf, sys->bus->data_in[1], "data_in1");
    sc_trace(tf, sys->bus->data_in[2], "data_in2");
    sc_trace(tf, sys->bus->data_in[3], "data_in3");
    sc_trace(tf, sys->bus->data_in[4], "data_in4");
    sc_trace(tf, sys->bus->data_in[5], "data_in5");
    sc_trace(tf, sys->bus->data_in[6], "data_in6");
    sc_trace(tf, sys->bus->data_in[7], "data_in7");
    sc_trace(tf, sys->bus->data_out[0], "data_out0");
    sc_trace(tf, sys->bus->data_out[1], "data_out1");
    sc_trace(tf, sys->bus->data_out[2], "data_out2");
    sc_trace(tf, sys->bus->data_out[3], "data_out3");
    sc_trace(tf, sys->bus->data_out[4], "data_out4");
    sc_trace(tf, sys->bus->data_out[5], "data_out5");
    sc_trace(tf, sys->bus->data_out[6], "data_out6");
    sc_trace(tf, sys->bus->data_out[7], "data_out7");
    sc_trace(tf, sys->bus->addr_in[0], "addr_in0");
    sc_trace(tf, sys->bus->addr_in[1], "addr_in1");
    sc_trace(tf, sys->bus->addr_in[2], "addr_in2");
```

```
    sc_trace(tf, sys->bus->addr_in[3], "addr_in3");
    sc_trace(tf, sys->bus->addr_in[4], "addr_in4");
    sc_trace(tf, sys->bus->addr_in[5], "addr_in5");
    sc_trace(tf, sys->bus->addr_in[6], "addr_in6");
    sc_trace(tf, sys->bus->addr_in[7], "addr_in7");
    sc_trace(tf, sys->bus->addr_out[0], "addr_out0");
    sc_trace(tf, sys->bus->addr_out[1], "addr_out1");
    sc_trace(tf, sys->bus->addr_out[2], "addr_out2");
    sc_trace(tf, sys->bus->addr_out[3], "addr_out3");
    sc_trace(tf, sys->bus->addr_out[4], "addr_out4");
    sc_trace(tf, sys->bus->addr_out[5], "addr_out5");
    sc_trace(tf, sys->bus->addr_out[6], "addr_out6");
    sc_trace(tf, sys->bus->addr_out[7], "addr_out7");

    // run simulation
    sc_start(16000*20);

    sc_close_vcd_trace_file(tf);

    return(0);
}
```

A.18 network_bus.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : network_bus.cpp                          **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                  **
 ** Date             : 29.12.2002                               **
 **                                                             **
 ** Function         : Connects wrapped cores with bus nodes.   **
 ** Notes            : Connects 3 FIR cores, 3 IIR cores, 1 input **
 **                  : core and 1 output core in a bus structure. **
 **                  : Can output run-time report and traffic   **
 **                  : statistics to files.                     **
 **                                                             **
 ** Major revisions : 26.12.2002   All cores added to bus.     **
 **                  : 29.12.2002   Report and statistics generation **
 **                  :              added.                      **
 **                                                             **
 *****************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"
#include "ocp_bus_node.cpp"
#include "fir_ocp_wrap.cpp"
#include "iir_ocp_wrap.cpp"
#include "input_ocp_wrap.cpp"
#include "output_ocp_wrap.cpp"

template< int FIR_COUNT, int IIR_COUNT, int DATA_WIDTH, int ADDR_WIDTH, const char *
    imem_name1, const char *cmem_name1, const char *fmem_name1, const char *imem_name2,
    const char *cmem_name2, const char *fmem_name2, const char *imem_name3, const char *
    cmem_name3, const char *fmem_name3, const char *imem_name4, const char *cmem_name4,
    const char *imem_name5, const char *cmem_name5, const char *imem_name6, const char *
    cmem_name6, const char *input_name, const char *output_name, bool STATISTICS_ON >
SC_MODULE(network_bus) {
```

```
  /*********************************
  **          Ports                **
  *********************************/

  sc_in_clk                        clk;
  sc_in< bool >                    nrst;            // active low, synchronous reset

40
  /*********************************
  **          Signals              **
  *********************************/

//    sc_signal< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > send[FIR_COUNT+IIR_COUNT+2];     //
    packet from interface
//    sc_signal< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > recv[FIR_COUNT+IIR_COUNT+2] ;
      // packet to interface
//    sc_signal< bool > send_hold[FIR_COUNT+IIR_COUNT+2];        // send hold to
    interface
//    sc_signal< bool > recv_hold[FIR_COUNT+IIR_COUNT+2];        // receive hold from
    interface
//    sc_signal< bool > send_req[FIR_COUNT+IIR_COUNT+2];         // send request from
    interface
//    sc_signal< bool > recv_req[FIR_COUNT+IIR_COUNT+2];         // receive request to
    interface
50

  sc_signal_rv<ADDR_WIDTH> addr;                                 // address bus
  sc_signal_rv<DATA_WIDTH> data;                                 // data bus
  sc_signal< bool > hold_out[FIR_COUNT+IIR_COUNT+2];             // hold the entire bus
  sc_signal< bool > hold_in;                                     // ...
  sc_signal< bool > token_ring[FIR_COUNT+IIR_COUNT+2];           // bus permit token input
  sc_signal< bool > req_out[FIR_COUNT+IIR_COUNT+2];              // indicate valid data on
      bus
  sc_signal< bool > req_in;                                      // ...
  sc_signal< sc_uint<3> > nodeM_MCmd[FIR_COUNT+IIR_COUNT+2];             // OCP signals
      node master
60
  sc_signal< sc_uint<CHANNEL_WIDTH> > nodeM_MAddr[FIR_COUNT+IIR_COUNT+2]; // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_MData[FIR_COUNT+IIR_COUNT+2];     // ...
  sc_signal< bool > nodeM_SCmdAccept[FIR_COUNT+IIR_COUNT+2];              // ...
  sc_signal< sc_int<DATA_WIDTH> > nodeM_SData[FIR_COUNT+IIR_COUNT+2];     // ...
  sc_signal< sc_uint<2> > nodeM_SResp[FIR_COUNT+IIR_COUNT+2];             // ...
  sc_signal< sc_uint<3> > intM_MCmd[FIR_COUNT+IIR_COUNT+2];               // OCP signals
      interface master
  sc_signal< sc_uint<ADDR_WIDTH> > intM_MAddr[FIR_COUNT+IIR_COUNT+2];   // ...
  sc_signal< sc_int<DATA_WIDTH> > intM_MData[FIR_COUNT+IIR_COUNT+2];      // ...
  sc_signal< bool > intM_SCmdAccept[FIR_COUNT+IIR_COUNT+2];               // ...
  sc_signal< sc_int<DATA_WIDTH> > intM_SData[FIR_COUNT+IIR_COUNT+2];      // ...
70
  sc_signal< sc_uint<2> > intM_SResp[FIR_COUNT+IIR_COUNT+2];              // ...

  /*********************************
  **          Variables            **
  *********************************/

  bool req_in_int;
  bool hold_in_int;
//    bool send_hold_in_int;
  long cc_count;    // Clock cycle count
80
  int i;
  int hold_no;      // counters
  int req_no;       // ...
//    int send_hold_no; // ...
  int stat[FIR_COUNT+IIR_COUNT+2][3]; // stores request and hold counts for cores
  FILE *p_report;
```

```
    FILE *p_stat;

    /*********************************
    **          Constants            **
    *********************************/

//        char message[] = "Run-time report for simulation of mini-core system.\nShows:\n-Clock cycle\n-Network holds, node with full buffer stalling entire network\n-Send holds, node waiting to send but not having permission.\nFor each is shown the address of the node responsible for the signal. 99 indicates that no node is setting the signal\n\nClock cycle\tNetwork hold\tReq from node\tSend hold\n");

    /*********************************
    **          Components           **
    *********************************/

    input_ocp_wrap<10, DATA_WIDTH, ADDR_WIDTH, input_name> *input_wrap1;
    output_ocp_wrap<6, DATA_WIDTH, ADDR_WIDTH, output_name> *output_wrap1;

    fir_ocp_wrap<0, DATA_WIDTH, ADDR_WIDTH, imem_name1, cmem_name1, fmem_name1> *fir_wrap1;
    fir_ocp_wrap<1, DATA_WIDTH, ADDR_WIDTH, imem_name2, cmem_name2, fmem_name2> *fir_wrap2;
    fir_ocp_wrap<2, DATA_WIDTH, ADDR_WIDTH, imem_name3, cmem_name3, fmem_name3> *fir_wrap3;
    iir_ocp_wrap<3, DATA_WIDTH, ADDR_WIDTH, imem_name4, cmem_name4> *iir_wrap1;
    iir_ocp_wrap<4, DATA_WIDTH, ADDR_WIDTH, imem_name5, cmem_name5> *iir_wrap2;
    iir_ocp_wrap<5, DATA_WIDTH, ADDR_WIDTH, imem_name6, cmem_name6> *iir_wrap3;

    ocp_bus_node<10, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 1> *node1;
    ocp_bus_node<0, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node2;
    ocp_bus_node<1, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node3;
    ocp_bus_node<2, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node4;
    ocp_bus_node<3, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node5;
    ocp_bus_node<4, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node6;
    ocp_bus_node<5, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node7;
    ocp_bus_node<6, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> *node8;


    /*********************************
    **         Constructor           **
    *********************************/

    SC_CTOR(network_bus) {

        char message[] = "Run-time report for simulation of mini-core system.\nShows:\n-Clock cycle\n-Network holds, node with full buffer stalling entire network\n-Send holds, node waiting to send but not having permission.\nFor each is shown the address of the node responsible for the signal. 99 indicates that no node is setting the signal\n\nClock cycle\tNetwork hold\tReq from node\tSend hold\n";

      // methods

      SC_METHOD( OR_N );        // ORs N inputs
      for(i=0; i<FIR_COUNT+IIR_COUNT+2; i++)
        sensitive << req_out[i] << hold_out[i];

      SC_METHOD( statistics ); // Creates run-time report and statistics
      sensitive_pos << clk;

      // input core

      input_wrap1 = new input_ocp_wrap<10, DATA_WIDTH, ADDR_WIDTH, input_name> ("
          input_wrap1");
```

```
        input_wrap1->clk(clk);
        input_wrap1->nrst(nrst);
        input_wrap1->M_MCmd(intM_MCmd[0]);
        input_wrap1->M_MAddr(intM_MAddr[0]);
        input_wrap1->M_MData(intM_MData[0]);
        input_wrap1->M_SCmdAccept(intM_SCmdAccept[0]);
        input_wrap1->M_SData(intM_SData[0]);
        input_wrap1->M_SResp(intM_SResp[0]);
        input_wrap1->S_MCmd(nodeM_MCmd[0]);
        input_wrap1->S_MAddr(nodeM_MAddr[0]);
        input_wrap1->S_MData(nodeM_MData[0]);
        input_wrap1->S_SCmdAccept(nodeM_SCmdAccept[0]);
        input_wrap1->S_SData(nodeM_SData[0]);
        input_wrap1->S_SResp(nodeM_SResp[0]);


        // output core

        output_wrap1 = new output_ocp_wrap<6, DATA_WIDTH, ADDR_WIDTH, output_name>("
            output_wrap1");
        output_wrap1->clk(clk);
        output_wrap1->nrst(nrst);
        output_wrap1->M_MCmd(intM_MCmd[7]);
        output_wrap1->M_MAddr(intM_MAddr[7]);
        output_wrap1->M_MData(intM_MData[7]);
        output_wrap1->M_SCmdAccept(intM_SCmdAccept[7]);
        output_wrap1->M_SData(intM_SData[7]);
        output_wrap1->M_SResp(intM_SResp[7]);
        output_wrap1->S_MCmd(nodeM_MCmd[7]);
        output_wrap1->S_MAddr(nodeM_MAddr[7]);
        output_wrap1->S_MData(nodeM_MData[7]);
        output_wrap1->S_SCmdAccept(nodeM_SCmdAccept[7]);
        output_wrap1->S_SData(nodeM_SData[7]);
        output_wrap1->S_SResp(nodeM_SResp[7]);

        // FIR cores with interface

        fir_wrap1 = new fir_ocp_wrap<0, DATA_WIDTH, ADDR_WIDTH, imem_name1, cmem_name1,
            fmem_name1>("fir_wrap1");
        fir_wrap1->clk(clk);
        fir_wrap1->nrst(nrst);
        fir_wrap1->M_MCmd(intM_MCmd[1]);
        fir_wrap1->M_MAddr(intM_MAddr[1]);
        fir_wrap1->M_MData(intM_MData[1]);
        fir_wrap1->M_SCmdAccept(intM_SCmdAccept[1]);
        fir_wrap1->M_SData(intM_SData[1]);
        fir_wrap1->M_SResp(intM_SResp[1]);
        fir_wrap1->S_MCmd(nodeM_MCmd[1]);
        fir_wrap1->S_MAddr(nodeM_MAddr[1]);
        fir_wrap1->S_MData(nodeM_MData[1]);
        fir_wrap1->S_SCmdAccept(nodeM_SCmdAccept[1]);
        fir_wrap1->S_SData(nodeM_SData[1]);
        fir_wrap1->S_SResp(nodeM_SResp[1]);

        fir_wrap2 = new fir_ocp_wrap<1, DATA_WIDTH, ADDR_WIDTH, imem_name2, cmem_name2,
            fmem_name2>("fir_wrap2");
        fir_wrap2->clk(clk);
        fir_wrap2->nrst(nrst);
        fir_wrap2->M_MCmd(intM_MCmd[2]);
        fir_wrap2->M_MAddr(intM_MAddr[2]);
        fir_wrap2->M_MData(intM_MData[2]);
        fir_wrap2->M_SCmdAccept(intM_SCmdAccept[2]);
```

```
        fir_wrap2->M_SData(intM_SData[2]);
        fir_wrap2->M_SResp(intM_SResp[2]);
        fir_wrap2->S_MCmd(nodeM_MCmd[2]);
        fir_wrap2->S_MAddr(nodeM_MAddr[2]);
        fir_wrap2->S_MData(nodeM_MData[2]);
        fir_wrap2->S_SCmdAccept(nodeM_SCmdAccept[2]);
        fir_wrap2->S_SData(nodeM_SData[2]);
        fir_wrap2->S_SResp(nodeM_SResp[2]);

        fir_wrap3 = new fir_ocp_wrap<2, DATA_WIDTH, ADDR_WIDTH, imem_name3, cmem_name3,
            fmem_name3> ("fir_wrap3");
        fir_wrap3->clk(clk);
        fir_wrap3->nrst(nrst);
        fir_wrap3->M_MCmd(intM_MCmd[3]);
        fir_wrap3->M_MAddr(intM_MAddr[3]);
        fir_wrap3->M_MData(intM_MData[3]);
        fir_wrap3->M_SCmdAccept(intM_SCmdAccept[3]);
        fir_wrap3->M_SData(intM_SData[3]);
        fir_wrap3->M_SResp(intM_SResp[3]);
        fir_wrap3->S_MCmd(nodeM_MCmd[3]);
        fir_wrap3->S_MAddr(nodeM_MAddr[3]);
        fir_wrap3->S_MData(nodeM_MData[3]);
        fir_wrap3->S_SCmdAccept(nodeM_SCmdAccept[3]);
        fir_wrap3->S_SData(nodeM_SData[3]);
        fir_wrap3->S_SResp(nodeM_SResp[3]);


        // IIR cores with interface

        iir_wrap1 = new iir_ocp_wrap<3, DATA_WIDTH, ADDR_WIDTH, imem_name4, cmem_name4> ("
            iir_wrap1");
        iir_wrap1->clk(clk);
        iir_wrap1->nrst(nrst);
        iir_wrap1->M_MCmd(intM_MCmd[4]);
        iir_wrap1->M_MAddr(intM_MAddr[4]);
        iir_wrap1->M_MData(intM_MData[4]);
        iir_wrap1->M_SCmdAccept(intM_SCmdAccept[4]);
        iir_wrap1->M_SData(intM_SData[4]);
        iir_wrap1->M_SResp(intM_SResp[4]);
        iir_wrap1->S_MCmd(nodeM_MCmd[4]);
        iir_wrap1->S_MAddr(nodeM_MAddr[4]);
        iir_wrap1->S_MData(nodeM_MData[4]);
        iir_wrap1->S_SCmdAccept(nodeM_SCmdAccept[4]);
        iir_wrap1->S_SData(nodeM_SData[4]);
        iir_wrap1->S_SResp(nodeM_SResp[4]);

        iir_wrap2 = new iir_ocp_wrap<4, DATA_WIDTH, ADDR_WIDTH, imem_name5, cmem_name5> ("
            iir_wrap2");
        iir_wrap2->clk(clk);
        iir_wrap2->nrst(nrst);
        iir_wrap2->M_MCmd(intM_MCmd[5]);
        iir_wrap2->M_MAddr(intM_MAddr[5]);
        iir_wrap2->M_MData(intM_MData[5]);
        iir_wrap2->M_SCmdAccept(intM_SCmdAccept[5]);
        iir_wrap2->M_SData(intM_SData[5]);
        iir_wrap2->M_SResp(intM_SResp[5]);
        iir_wrap2->S_MCmd(nodeM_MCmd[5]);
        iir_wrap2->S_MAddr(nodeM_MAddr[5]);
        iir_wrap2->S_MData(nodeM_MData[5]);
        iir_wrap2->S_SCmdAccept(nodeM_SCmdAccept[5]);
        iir_wrap2->S_SData(nodeM_SData[5]);
        iir_wrap2->S_SResp(nodeM_SResp[5]);
```

```
        iir_wrap3 = new iir_ocp_wrap<5, DATA_WIDTH, ADDR_WIDTH, imem_name6, cmem_name6> ("
            iir_wrap3");
        iir_wrap3->clk(clk);
        iir_wrap3->nrst(nrst);
        iir_wrap3->M_MCmd(intM_MCmd[6]);
        iir_wrap3->M_MAddr(intM_MAddr[6]);
        iir_wrap3->M_MData(intM_MData[6]);
        iir_wrap3->M_SCmdAccept(intM_SCmdAccept[6]);
        iir_wrap3->M_SData(intM_SData[6]);
        iir_wrap3->M_SResp(intM_SResp[6]);
        iir_wrap3->S_MCmd(nodeM_MCmd[6]);
        iir_wrap3->S_MAddr(nodeM_MAddr[6]);
        iir_wrap3->S_MData(nodeM_MData[6]);
        iir_wrap3->S_SCmdAccept(nodeM_SCmdAccept[6]);
        iir_wrap3->S_SData(nodeM_SData[6]);
        iir_wrap3->S_SResp(nodeM_SResp[6]);


        // Bus nodes

        node1 = new ocp_bus_node<10, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 1> ("input_node")
            ;
        node1->clk(clk);
        node1->nrst(nrst);
        node1->addr(addr);
        node1->data(data);
        node1->hold_out(hold_out[0]);
        node1->hold_in(hold_in);
        node1->token_in(token_ring[0]);
        node1->token_out(token_ring[1]);
        node1->req_out(req_out[0]);
        node1->req_in(req_in);
        node1->M_MCmd(nodeM_MCmd[0]);
        node1->M_MAddr(nodeM_MAddr[0]);
        node1->M_MData(nodeM_MData[0]);
        node1->M_SCmdAccept(nodeM_SCmdAccept[0]);
        node1->M_SData(nodeM_SData[0]);
        node1->M_SResp(nodeM_SResp[0]);
        node1->S_MCmd(intM_MCmd[0]);
        node1->S_MAddr(intM_MAddr[0]);
        node1->S_MData(intM_MData[0]);
        node1->S_SCmdAccept(intM_SCmdAccept[0]);
        node1->S_SData(intM_SData[0]);
        node1->S_SResp(intM_SResp[0]);

        node2 = new ocp_bus_node<0, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("FIR1_node");
        node2->clk(clk);
        node2->nrst(nrst);
        node2->addr(addr);
        node2->data(data);
        node2->hold_out(hold_out[1]);
        node2->hold_in(hold_in);
        node2->token_in(token_ring[1]);
        node2->token_out(token_ring[2]);
        node2->req_out(req_out[1]);
        node2->req_in(req_in);
        node2->M_MCmd(nodeM_MCmd[1]);
        node2->M_MAddr(nodeM_MAddr[1]);
        node2->M_MData(nodeM_MData[1]);
        node2->M_SCmdAccept(nodeM_SCmdAccept[1]);
        node2->M_SData(nodeM_SData[1]);
```

```
        node2->M_SResp(nodeM_SResp[1]);
        node2->S_MCmd(intM_MCmd[1]);
        node2->S_MAddr(intM_MAddr[1]);
        node2->S_MData(intM_MData[1]);
320     node2->S_SCmdAccept(intM_SCmdAccept[1]);
        node2->S_SData(intM_SData[1]);
        node2->S_SResp(intM_SResp[1]);

        node3 = new ocp_bus_node<1, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("FIR2_node");
        node3->clk(clk);
        node3->nrst(nrst);
        node3->addr(addr);
        node3->data(data);
        node3->hold_out(hold_out[2]);
330     node3->hold_in(hold_in);
        node3->token_in(token_ring[2]);
        node3->token_out(token_ring[3]);
        node3->req_out(req_out[2]);
        node3->req_in(req_in);
        node3->M_MCmd(nodeM_MCmd[2]);
        node3->M_MAddr(nodeM_MAddr[2]);
        node3->M_MData(nodeM_MData[2]);
        node3->M_SCmdAccept(nodeM_SCmdAccept[2]);
        node3->M_SData(nodeM_SData[2]);
340     node3->M_SResp(nodeM_SResp[2]);
        node3->S_MCmd(intM_MCmd[2]);
        node3->S_MAddr(intM_MAddr[2]);
        node3->S_MData(intM_MData[2]);
        node3->S_SCmdAccept(intM_SCmdAccept[2]);
        node3->S_SData(intM_SData[2]);
        node3->S_SResp(intM_SResp[2]);

        node4 = new ocp_bus_node<2, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("FIR3_node");
        node4->clk(clk);
350     node4->nrst(nrst);
        node4->addr(addr);
        node4->data(data);
        node4->hold_out(hold_out[3]);
        node4->hold_in(hold_in);
        node4->token_in(token_ring[3]);
        node4->token_out(token_ring[4]);
        node4->req_out(req_out[3]);
        node4->req_in(req_in);
        node4->M_MCmd(nodeM_MCmd[3]);
360     node4->M_MAddr(nodeM_MAddr[3]);
        node4->M_MData(nodeM_MData[3]);
        node4->M_SCmdAccept(nodeM_SCmdAccept[3]);
        node4->M_SData(nodeM_SData[3]);
        node4->M_SResp(nodeM_SResp[3]);
        node4->S_MCmd(intM_MCmd[3]);
        node4->S_MAddr(intM_MAddr[3]);
        node4->S_MData(intM_MData[3]);
        node4->S_SCmdAccept(intM_SCmdAccept[3]);
        node4->S_SData(intM_SData[3]);
370     node4->S_SResp(intM_SResp[3]);


        node5 = new ocp_bus_node<3, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("IIR1_node");
        node5->clk(clk);
        node5->nrst(nrst);
        node5->addr(addr);
        node5->data(data);
```

```
        node5->hold_out(hold_out[4]);
        node5->hold_in(hold_in);
        node5->token_in(token_ring[4]);
        node5->token_out(token_ring[5]);
        node5->req_out(req_out[4]);
        node5->req_in(req_in);
        node5->M_MCmd(nodeM_MCmd[4]);
        node5->M_MAddr(nodeM_MAddr[4]);
        node5->M_MData(nodeM_MData[4]);
        node5->M_SCmdAccept(nodeM_SCmdAccept[4]);
        node5->M_SData(nodeM_SData[4]);
        node5->M_SResp(nodeM_SResp[4]);
        node5->S_MCmd(intM_MCmd[4]);
        node5->S_MAddr(intM_MAddr[4]);
        node5->S_MData(intM_MData[4]);
        node5->S_SCmdAccept(intM_SCmdAccept[4]);
        node5->S_SData(intM_SData[4]);
        node5->S_SResp(intM_SResp[4]);

        node6 = new ocp_bus_node<4, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("IIR2_node");
        node6->clk(clk);
        node6->nrst(nrst);
        node6->addr(addr);
        node6->data(data);
        node6->hold_out(hold_out[5]);
        node6->hold_in(hold_in);
        node6->token_in(token_ring[5]);
        node6->token_out(token_ring[6]);
        node6->req_out(req_out[5]);
        node6->req_in(req_in);
        node6->M_MCmd(nodeM_MCmd[5]);
        node6->M_MAddr(nodeM_MAddr[5]);
        node6->M_MData(nodeM_MData[5]);
        node6->M_SCmdAccept(nodeM_SCmdAccept[5]);
        node6->M_SData(nodeM_SData[5]);
        node6->M_SResp(nodeM_SResp[5]);
        node6->S_MCmd(intM_MCmd[5]);
        node6->S_MAddr(intM_MAddr[5]);
        node6->S_MData(intM_MData[5]);
        node6->S_SCmdAccept(intM_SCmdAccept[5]);
        node6->S_SData(intM_SData[5]);
        node6->S_SResp(intM_SResp[5]);

        node7 = new ocp_bus_node<5, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("IIR3_node");
        node7->clk(clk);
        node7->nrst(nrst);
        node7->addr(addr);
        node7->data(data);
        node7->hold_out(hold_out[6]);
        node7->hold_in(hold_in);
        node7->token_in(token_ring[6]);
        node7->token_out(token_ring[7]);
        node7->req_out(req_out[6]);
        node7->req_in(req_in);
        node7->M_MCmd(nodeM_MCmd[6]);
        node7->M_MAddr(nodeM_MAddr[6]);
        node7->M_MData(nodeM_MData[6]);
        node7->M_SCmdAccept(nodeM_SCmdAccept[6]);
        node7->M_SData(nodeM_SData[6]);
        node7->M_SResp(nodeM_SResp[6]);
        node7->S_MCmd(intM_MCmd[6]);
        node7->S_MAddr(intM_MAddr[6]);
```

```
440       node7->S_MData(intM_MData[6]);
          node7->S_SCmdAccept(intM_SCmdAccept[6]);
          node7->S_SData(intM_SData[6]);
          node7->S_SResp(intM_SResp[6]);

          node8 = new ocp_bus_node<6, DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH, 0> ("output_node")
              ;
          node8->clk(clk);
          node8->nrst(nrst);
          node8->addr(addr);
          node8->data(data);
450       node8->hold_out(hold_out[7]);
          node8->hold_in(hold_in);
          node8->token_in(token_ring[7]);
          node8->token_out(token_ring[0]);
          node8->req_out(req_out[7]);
          node8->req_in(req_in);
          node8->M_MCmd(nodeM_MCmd[7]);
          node8->M_MAddr(nodeM_MAddr[7]);
          node8->M_MData(nodeM_MData[7]);
          node8->M_SCmdAccept(nodeM_SCmdAccept[7]);
460       node8->M_SData(nodeM_SData[7]);
          node8->M_SResp(nodeM_SResp[7]);
          node8->S_MCmd(intM_MCmd[7]);
          node8->S_MAddr(intM_MAddr[7]);
          node8->S_MData(intM_MData[7]);
          node8->S_SCmdAccept(intM_SCmdAccept[7]);
          node8->S_SData(intM_SData[7]);
          node8->S_SResp(intM_SResp[7]);

          // open files
470       if (STATISTICS_ON) {
            if ( (p_report = fopen("report.txt", "w")) == NULL)  {
              printf("Error creating report file !!!\n");
              abort();
            }
            fprintf(p_report, message);
            //         fprintf(p_report, "Clock cycle\thold from node\treq from node\n");
            //       if ( (p_stat = fopen("stat.txt", "w")) == NULL)  {
            //         printf("Error creating statistics file !!!\n");
            //         abort();
480         //       }
          }
        } // end constructor

        void statistics() {

          if (nrst == 0) {
            cc_count = 0;
            for (i=0;i<IIR_COUNT+FIR_COUNT+2;i++) {
                stat[i][0] = 0;
490             stat[i][1] = 0;
            }
          }
          else {
            if (STATISTICS_ON) {
              if (hold_in || req_in) {  // bus transaction this CC

                // create report
                hold_no = 99;    // ...
                req_no = 99;     // 99 indicates no active req or hold
500 //          send_hold_no = 99;
```

```
            for (i=0; i<FIR_COUNT+IIR_COUNT+2;i++) {
              if (hold_out[i])
                hold_no = i;
              if (req_out[i])
                req_no = i;
//            if (send_hold[i])
//              send_hold_no = i;
            }
            fprintf(p_report, "%u\t\t%u\t\t%u\n", cc_count, hold_no, req_no);
            fflush(p_report);

            // create statistics
            if ( (p_stat = fopen("stat.txt", "w")) == NULL)  {
              printf("Error creating statistics file!!!\n");
              abort();
            }
            for (i=0; i<FIR_COUNT+IIR_COUNT+2;i++) {
              if (hold_out[i])
                stat[i][0]++;
              if (req_out[i])
                stat[i][1]++;
//            if (send_hold[i])
//              stat[i][2]++;
            }
            for (i=0; i<FIR_COUNT+IIR_COUNT+2;i++) {
              fprintf(p_stat,"%u\n%u\n%u\n%u\n",i,stat[i][0],stat[i][1],stat[i][2]);
            }
            fclose(p_stat);
          }

        cc_count++;
        }
      }
    } // END OF statistics


    // ORs N inputs. Used for node requests and holds.
    void OR_N() {

      int i;
      hold_in_int = 0;
      req_in_int = 0;
//      send_hold_in_int = 0;

      for (i=0; i<FIR_COUNT+IIR_COUNT+2; i++) {
        hold_in_int = hold_in_int | hold_out[i];
        req_in_int = req_in_int | req_out[i];
//        send_hold_in_int = send_hold_in_int | send_hold[i];
      }
      req_in = req_in_int;
      hold_in = hold_in_int;
//      send_hold_in = send_hold_in_int;

    } // END OF OR_N



}; // END OF network_bus
```

A.19 ocp_bus_node.cpp

```
/*****************************************************************
 **                                                             **
 ** Name             : ocp_bus_node.cpp                        **
 **                                                             **
 ** Author           : Karsten Larsen, c971833                 **
 ** Date             : 19.12.2002                              **
 **                                                             **
 ** Function         : Bus node with ocp interface.            **
 **                  : Handles bus transaction - including local **
 **                  : arbiting.                               **
 ** Notes            : Functions both as OCP master and slave. **
 **                  : Token passing is initiated each rising clock **
 **                  : edge and token traverses combinatorial through **
 **                  : the nodes.                              **
 **                  : Based upon bus_node.cpp.                **
 **                                                             **
 ** Major revisions : 06.01.2003   Final version.             **
 **                                                             **
 *****************************************************************/

#include "systemc.h"
#include "ocp_dec.h"

template<int ADDRESS, int DATA_WDTH, int S_ADDR_WDTH, int M_ADDR_WDTH, bool
    token_initiator>
SC_MODULE(ocp_bus_node) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;            // active low, synchronous reset
  sc_out< sc_uint<3> >               M_MCmd;          // OCP signals as master
  sc_out< sc_uint<M_ADDR_WDTH> >     M_MAddr;         // ...
  sc_out< sc_int<DATA_WDTH> >        M_MData;         // ...
  sc_in< bool >                      M_SCmdAccept;    // ...
  sc_in< sc_int<DATA_WDTH> >         M_SData;         // ...
  sc_in< sc_uint<2> >                M_SResp;         // ...
  sc_in< sc_uint<3> >                S_MCmd;          // OCP signals as slave
  sc_in< sc_uint<S_ADDR_WDTH> >      S_MAddr;         // ...
  sc_in< sc_int<DATA_WDTH> >         S_MData;         // ...
  sc_out< bool >                     S_SCmdAccept;    // ...
  sc_out< sc_int<DATA_WDTH> >        S_SData;         // ...
  sc_out< sc_uint<2> >               S_SResp;         // ...
  sc_inout_rv<S_ADDR_WDTH>           addr;            // address bus
  sc_inout_rv<DATA_WDTH>             data;            // data bus
  sc_out< bool >                     hold_out;        // hold the entire bus
  sc_in< bool >                      hold_in;         // ...
  sc_in< bool >                      token_in;        // bus permit token input
  sc_out< bool >                     token_out;       // bus permit token output
  sc_out< bool >                     req_out;         // indicate valid data on bus
  sc_out< bool >                     req_in;          // ...


  /*********************************
  **          Variables           **
  *********************************/

  bool token_initiate;
  bool write_to_bus;
  bool recv_ready;
```

```
    bool trig_process;
    bool token_int;
    bool send_ready;
    bool pending_request;
    int first_run;
    int M_State;
    int S_State;
    int wait_cc;
70  unsigned int INT_ADDRESS;
    unsigned int INT_ADDRESS_TOP;
    unsigned int int_addr;
    sc_uint<DATA_WDTH> internal_recv_data;
    sc_uint<M_ADDR_WDTH> internal_recv_addr;
    sc_uint<DATA_WDTH> internal_send_data;
    sc_uint<S_ADDR_WDTH> internal_send_addr;
    sc_uint<S_ADDR_WDTH+DATA_WDTH+1> temp;
    sc_lv<M_ADDR_WDTH> addr_hi_z;
    sc_lv<DATA_WDTH> data_hi_z;
80  sc_uint<S_ADDR_WDTH> temp_addr;


    /*********************************
    **        Constructor           **
    *********************************/

    SC_CTOR(ocp_bus_node) {

      // create high impedance constants
90    int i;
      for (i=0; i<S_ADDR_WDTH; i++)
        addr_hi_z[i] = 'Z';
      for (i=0; i<DATA_WDTH; i++)
        data_hi_z[i] = 'Z';

      token_initiate = token_initiator;  // indicates whether node is to initiate token
          passing

      SC_METHOD( ocp_master );
      sensitive_pos << clk;
100
      SC_METHOD( ocp_slave );
      sensitive_pos << clk;

      SC_METHOD( bus_read );
      sensitive_pos << clk;

      SC_METHOD( bus_write );
      sensitive_neg << clk;

110   SC_METHOD( token_handle );
      sensitive << token_in;

      SC_METHOD( token_init );
      sensitive_pos << clk;

      SC_METHOD( token_rst );
      sensitive_neg << clk;

    } // end constructor
120

    /*********************************
```

```
  **          bus_read()            **
  ** Reads bus on positive clk edge **
  ************************************/
void bus_read() {

  if (nrst.read() == 0) {                                          //active reset
    INT_ADDRESS = 4*ADDRESS;
    INT_ADDRESS_TOP = INT_ADDRESS+CHANNEL_COUNT-1;
    recv_ready = 0;
    hold_out.write(0);
  }
  else {                                                           // positive
      clockedge
    // receive - from network to interface
    if (req_in) {                                                  // valid data on
        bus
      if (recv_ready) {                                            // allready data
          in buffer
        hold_out.write(1);
      }
      else {                                                       // buffer empty
        hold_out.write(0);
        temp_addr = addr.read();                                   // ...
        int_addr = temp_addr;                                      // type
            conversion for comparison
        if (int_addr >= INT_ADDRESS && int_addr <= INT_ADDRESS_TOP) { //data for this
            node
          internal_recv_data = data.read();
          internal_recv_addr = addr.read();
          internal_recv_addr -= INT_ADDRESS;                       // map into
              internal core channel (0-3)
          recv_ready = 1;                                          // receive data
              ready
        }
      }
    }
  }
  return;
} // END OF node_read()


/************************************
  **          bus_write()            **
  ** Writes to bus negative clk edge **
  ************************************/
void bus_write() {

  if (nrst.read() == 0) {  //active reset
    addr.write(addr_hi_z);
    data.write(data_hi_z);
    token_initiate = token_initiator;
  }

  else {

    // send - from interface to network

    addr.write(addr_hi_z);
    data.write(data_hi_z);
    req_out.write(0);

    if (hold_in.read()) {                 // hold from network
```

```
      }
      else {  // no hold
        if (write_to_bus) {                // granted bus permission and data waiting
          req_out.write(1);                // bus data valid
          data.write(internal_send_data);
          addr.write(internal_send_addr);
          send_ready = 0;
        }
      }
    }
    return;
  } // END OF node_write()


  /*********************************
   **       token_handle()        **
   ** Handles token passing       **
   ** Combinatorial               **
   *********************************/
  void token_handle() {

    if (clk) {                     // only active when clock is high
      write_to_bus = 0;

      if (token_initiate) {        // token initiator, token cycle complete
        if (token_in) {            // received token
          token_initiate = 1;      // token initiator, next clockcycle
          if (send_ready) {        // waiting to transmit data
            write_to_bus = 1;      // ok to write to bus
          }
        }
        else {
          token_initiate = 0;      // new token initiator
        }
      }

      else {                       // not token initiator
        if (token_in) {            // received token

          if (send_ready) {        // waiting to transmit data
            token_out.write(0);    // do not pass token
            write_to_bus = 1;      // ok to write to bus
            token_initiate = 1;    // token initiator, next clockcycle
          }
          else {                   // no data
            token_out.write(1);    // no data, pass token
            token_initiate = 0;
          }
        }
        else {                     // no token
          token_out.write(0);
          token_initiate = 0;
        }
      }
    }
    return;
  } // END OF token_handle()


  /*********************************
   **        token_init()         **
   ** Initiates token chain,      **
```

```
240     ** if told so                    **
        ***********************************/
        void token_init (){

          if (nrst.read() == 0) {  //active reset
            write_to_bus = 0;
            token_out.write(0);
            first_run = 2;
          }
          else {
250         if (token_initiate) {  // token initiator
              token_out.write(1);  // pass token
              write_to_bus = 0;
            }
          }
          return;
        } // END OF token_init()


        /***********************************
260     **          token_rst()           **
        ** Reset token chain at falling **
        ** clock edge.                   **
        ***********************************/
        void token_rst() {

          // set token_initate if 0 on input at falling clockedge
          if (nrst.read() == 1) {
            if (first_run == 0) {
              if (token_initiate) {
270             if (token_in.read() == 0) {
                  token_initiate = 0;
                }
              }
            }
            else {
              first_run--;
            }

            // reset token ring
280         token_out.write(0);
          }
        } // END OF token_rst()


        /************************************
        **    SC_METHOD ocp_slave()          **
        ** OCP slave, interface is master.   **
        ************************************/

290     void ocp_slave() {

          if (nrst.read() == 0) {  //active reset

            send_ready = 0;
            S_State = IDLE;
            wait_cc = 0;
          }

          else {
300
            switch(S_State) {
```

```
      case IDLE:
        if (SgetRequest(0)) {                    // request from master, allways write
          if ( !send_ready ) {                   // buffer empty, store data
            internal_send_data = SgetData();
            internal_send_addr = S_Address();
            send_ready = 1;
            SRelease();
310         S_State = WAIT;
          }
        }
        break;
      case WAIT:                                 // wait 1 CC before processing next request
        S_State = IDLE;
        break;
      }
    }
    return;
320 } // END OF ocp_slave


/************************************
**    SC_METHOD ocp_master()        **
** OCP master, interface is slave.  **
************************************/

void ocp_master() {

330   if (nrst.read() == 0) {  //active reset

      M_State = IDLE;
      pending_request = 0;
    }

    else {

      switch(M_State) {

340   case IDLE:
        if (recv_ready) {
          if(!MgetSBusy()) {                     // slave not busy
            M_Address(internal_recv_addr);
            MputData(internal_recv_data);
            MputWriteRequest();
            recv_ready = 0;
            pending_request = 1;
            M_State = WAIT;
          }
350     }
        break;

      case WAIT:
        if (MgetResponse(0))                     // if slave response, no release necessary
          M_State = IDLE;
        break;

      }
      return;
360   }
} // END OF ocp_master
```

```
/*************************
** OCP master functions  **
*************************/

// set target address
void M_Address(sc_uint<S_ADDR_WDTH> Addr) {

  M_MAddr.write(Addr);
  return;

} // END OF M_Address


// set M_MData field
void MputData(sc_uint<DATA_WDTH> Data) {

  sc_int<DATA_WDTH> temp;
  temp = Data;
  M_MData.write(temp);
  return;

} // END OF M_MData


// check if slave is busy
bool MgetSBusy() {

  return pending_request;

} // END OF M_MgetSBusy


// issue write request
bool MputWriteRequest() {

  M_MCmd.write(WR);
  return pending_request;

} // END OF M_MputWriteRequest


// check if slave has responded
bool MgetResponse(bool Release) {

  if (M_SCmdAccept.read()) {
    M_MCmd.write(IDLE);
    pending_request = 0;
    return 1;
  }
  else
    return 0;

} // END OF MgetResponse


/*************************
**  OCP slave functions  **
*************************/

// return S_MAddr field
sc_uint<S_ADDR_WDTH> S_Address() {
```

```
    return S_MAddr.read();

  } // END OF S_Address

430

  // return S_MData field
  sc_int<DATA_WDTH> SgetData() {

    return S_MData.read();

  }

  // test if master has sent request
  bool SgetRequest(bool Release) {
440
    S_SCmdAccept.write(0); // default value

    int temp;
    temp  = S_MCmd.read();
    if (temp == WR) {      // only write is active
      if (Release)
        S_SCmdAccept.write(1);
      return 1;
    }
450 else
      return 0;

  } // END OF SgetRequest


  // acknowledge master request
  void SRelease() {

    S_SCmdAccept.write(1);
460 return;

  }

}; // END OF SC_MODULE
```

A.20 ocp_dec.h

```
/*********************************************************************
 **                                                                 **
 ** Name             : ocp_dec.h                                    **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 02.01.2003                                   **
 **                                                                 **
 ** Function         : OCP constants.                               **
 ** Major revisions : 02.01.2003   Final version                    **
10 **                                                                 **
 *********************************************************************/

// master commands

#define IDLE 0   // IDLE command and state
#define WR   1   // write
#define RD   2   // read
#define RDEX 3   // read exclusively
#define BCST 7   // broadcast
```

```
20

    // slave responses

    #define NULL 0    // no response
    #define DVA  1    // data valid/accept
    #define ERR  3    // error

    // FSM states

30  #define WAIT 1
```

A.21 ocp_generic_node.cpp

```
    /*********************************************************************
     **                                                                 **
     ** Name             : ocp_generic_node.cpp                        **
     **                                                                 **
     ** Author           : Karsten Larsen, c971833                     **
     ** Date             : 19.12.2002                                  **
     **                                                                 **
     ** Function         : Generic node with ocp interface.            **
     **                  : Independent of network structure.           **
10   ** Major revisions : 08.01.2002   First version.                  **
     **                                                                 **
     *********************************************************************/

    #ifndef _ocp_generic_node_cpp
    #define _ocp_generic_node_cpp

    #include "systemc.h"
    #include "ocp_dec.h"

20  template<int DATA_WDTH, int S_ADDR_WDTH, int M_ADDR_WDTH>
    SC_MODULE(ocp_generic_node) {

      /*********************************
      **          Ports               **
      *********************************/

      sc_in_clk                          clk;
      sc_in< bool >                      nrst;              // active low, synchronous reset
      sc_out< sc_uint<3> >               M_MCmd;            // OCP signals as master
30    sc_out< sc_uint<M_ADDR_WDTH> >     M_MAddr;           // ...
      sc_out< sc_int<DATA_WDTH> >        M_MData;           // ...
      sc_in< bool >                      M_SCmdAccept;      // ...
      sc_in< sc_int<DATA_WDTH> >         M_SData;           // ...
      sc_in< sc_uint<2> >                M_SResp;           // ...
      sc_in< sc_uint<3> >                S_MCmd;            // OCP signals as slave
      sc_in< sc_uint<S_ADDR_WDTH> >      S_MAddr;           // ...
      sc_in< sc_int<DATA_WDTH> >         S_MData;           // ...
      sc_out< bool >                     S_SCmdAccept;      // ...
      sc_out< sc_int<DATA_WDTH> >        S_SData;           // ...
40    sc_out< sc_uint<2> >               S_SResp;           // ...
      sc_out< sc_uint<DATA_WDTH> >       data_out;          // data to network
      sc_in< sc_uint<DATA_WDTH> >        data_in;           // data from network
      sc_out< bool >                     req_out;           // request to network
      sc_in< bool >                      req_in;            // request from network
      sc_out< sc_uint<S_ADDR_WDTH> >     addr_out;          // address to network
      sc_in< sc_uint<M_ADDR_WDTH> >      addr_in;           // address (channel) from network
      sc_in< bool >                      hold_in;           // hold from network
```

```
    sc_out< bool >                  hold_out;           // hold to network

50
    /*********************************
    **          Variables            **
    *********************************/

    bool recv_ready;
    bool send_ready;
    bool pending_request;
    int M_State;
    int S_State;
60  int channel;
    unsigned int int_addr;
    sc_uint<DATA_WDTH> internal_recv_data;
    sc_uint<M_ADDR_WDTH> internal_recv_addr;
    sc_uint<DATA_WDTH> internal_send_data;
    sc_uint<S_ADDR_WDTH> internal_send_addr;
    sc_uint<S_ADDR_WDTH+DATA_WDTH+1> temp;
    sc_uint<S_ADDR_WDTH> temp_addr;


70
    /*********************************
    **         Constructor           **
    *********************************/

    SC_CTOR(ocp_generic_node) {

      SC_METHOD( ocp_master );
      sensitive_pos << clk;

80    SC_METHOD( ocp_slave );
      sensitive_pos << clk;

      SC_METHOD( network_write );
      sensitive_neg << clk;

      SC_METHOD( network_read );
      sensitive_pos << clk;

    } // end constructor
90

    /***********************************
     **    SC_METHOD network_write()    **
     ** Sends data to network.         **
     ***********************************/

    void network_write() {

      if (nrst.read() == 0) {  //active reset
100     req_out.write(0);

      }
      else {
        if (!hold_in) {         // not stalled
          if (send_ready) {
            data_out.write(internal_send_data);
            addr_out.write(internal_send_addr);
            req_out.write(1);
            send_ready = 0;
```

```
110         }
            else {
              req_out.write(0);
            }
          }
        }
      } // END OF network_write()


      /***********************************
       **    SC_METHOD network_read()     **
       ** Reads data from network.        **
       ***********************************/

      void network_read() {
        if (nrst.read() == 0) {    //active reset
        }
        else {

          if (req_in.read()) {     // valid data
            if (recv_ready) {      // allready data in buffer
              hold_out.write(1);
            }
            else {                 // buffer empty, store
              hold_out.write(0);
              internal_recv_addr = addr_in.read();
              internal_recv_data = data_in.read();
              recv_ready = 1;
            }
          }
        }
      } // END OF network_read)()

      /***********************************
      **    SC_METHOD ocp_slave()         **
      ** OCP slave, interface is master.  **
      ***********************************/

      void ocp_slave() {

        if (nrst.read() == 0) {    //active reset
          send_ready = 0;
          S_State = IDLE;
        }

        else {

          switch(S_State) {

          case IDLE:
            if (SgetRequest(0)) {  // request from master, allways write
              if ( !send_ready ) { // buffer empty, store data
                internal_send_data = SgetData();
                internal_send_addr = S_Address();
                send_ready = 1;
                SRelease();
                S_State = WAIT;
              }
            }
            break;
          case WAIT:               // wait 1 CC before processing next request
            S_State = IDLE;
```

```
        break;
      }
    }
    return;
  } // END OF ocp_slave


  /************************************
  **    SC_METHOD ocp_master()        **
  ** OCP master, interface is slave.  **
  ************************************/

  void ocp_master() {

    if (nrst.read() == 0) {  //active reset

      M_State = IDLE;
      pending_request = 0;
    }

    else {

      switch(M_State) {

      case IDLE:
        if (recv_ready) {
          if(!MgetSBusy()) {                  // slave not busy
            M_Address(internal_recv_addr);
            MputData(internal_recv_data);
            MputWriteRequest();
            recv_ready = 0;
            pending_request = 1;
            M_State = WAIT;
          }
        }
        break;

      case WAIT:
        if (MgetResponse(0))                  // if slave response, no release necessary
          M_State = IDLE;
        break;

      }
      return;
    }
  } // END OF ocp_master


  /*************************
  ** OCP master functions  **
  *************************/

  // set target address
  void M_Address(sc_uint<S_ADDR_WDTH> Addr) {

    M_MAddr.write(Addr);
    return;

  } // END OF M_Address


  // set M_MData field
```

```
  void MputData(sc_uint<DATA_WDTH> Data) {

    sc_int<DATA_WDTH> temp;
    temp = Data;
    M_MData.write(temp);
    return;

  } // END OF M_MData


  // check if slave is busy
  bool MgetSBusy() {

    return pending_request;

  } // END OF M_MgetSBusy


  // issue write request
  bool MputWriteRequest() {

    M_MCmd.write(WR);
    return pending_request;

  } // END OF M_MputWriteRequest


  // check if slave has responded
  bool MgetResponse(bool Release) {

    if (M_SCmdAccept.read()) {
      M_MCmd.write(IDLE);
      pending_request = 0;
      return 1;
    }
    else
      return 0;

  } // END OF MgetResponse


  /*************************
  **  OCP slave functions  **
  *************************/

  // return S_MAddr field
  sc_uint<S_ADDR_WDTH> S_Address() {

    return S_MAddr.read();

  } // END OF S_Address


  // return S_MData field
  sc_int<DATA_WDTH> SgetData() {

    return S_MData.read();

  }

  // test if master has sent request
  bool SgetRequest(bool Release) {
```

```
      S_SCmdAccept.write(0);              // default value

      int temp;
300   temp   = S_MCmd.read();
      if (temp == WR) {                   // only write is active
        if (Release)
          S_SCmdAccept.write(1);
        return 1;
      }
      else
        return 0;

    } // END OF SgetRequest
310

    // acknowledge master request
    void SRelease() {

      S_SCmdAccept.write(1);
      return;

    }

320 }; // END OF SC_MODULE

#endif
```

A.22 ocp_interface.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : ocp_interface.cpp                            **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 29.11.2002                                   **
 **                                                                 **
 ** Function         : OCP Communications interface for the         **
 **                  : mini-cores.                                  **
10 ** Notes            : Core communication same as in                **
 **                  : simple_interface.cpp.                        **
 ** Major revisions : 30.12.2002   Final version                    **
 **                                                                 **
 *********************************************************************/

#ifndef _ocp_interface_cpp
#define _ocp_interface_cpp

#include "systemc.h"
20 #include "ocp_dec.h"

template< int ADDRESS, int DATA_WDTH, int S_ADDR_WDTH, int M_ADDR_WDTH >
SC_MODULE(ocp_interface) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                       clk;
30 sc_in< bool >                   nrst;          // active low, synchronous reset
  sc_in< bool >                   core_req;      // request from core
```

```
    sc_in< bool >                    core_rw;          // read/write from core
    sc_in< sc_uint<S_ADDR_WDTH> >    core_read;        // receive address from core
    sc_out< sc_uint<DATA_WDTH> >     core_data_out;    // data from core
    sc_in< sc_uint<M_ADDR_WDTH> >    core_write;       // send address from core
    sc_in< sc_uint<DATA_WDTH> >      core_data_in;     // data to core
    sc_out< bool >                   core_hold;        // hold to core
    sc_out< sc_uint<3> >             M_MCmd;           // OCP signals as master
    sc_out< sc_uint<M_ADDR_WDTH> >   M_MAddr;          // ...
    sc_out< sc_int<DATA_WDTH> >      M_MData;          // ...
    sc_in< bool >                    M_SCmdAccept;     // ...
    sc_in< sc_int<DATA_WDTH> >       M_SData;          // ...
    sc_in< sc_uint<2> >              M_SResp;          // ...
    sc_in< sc_uint<3> >              S_MCmd;           // OCP signals as slave
    sc_in< sc_uint<S_ADDR_WDTH> >    S_MAddr;          // ...
    sc_in< sc_int<DATA_WDTH> >       S_MData;          // ...
    sc_out< bool >                   S_SCmdAccept;     // ...
    sc_out< sc_int<DATA_WDTH> >      S_SData;          // ...
    sc_out< sc_uint<2> >             S_SResp;          // ...

    /*********************************
    **          Registers           **
    *********************************/

    sc_uint<DATA_WDTH+1> packet_recv[CHANNEL_COUNT];  //buffered receive packets, N
        channels, tag(1):addr:data
    sc_uint<M_ADDR_WDTH+DATA_WDTH+1> packet_send;     //buffered send packet, tag(1):addr:
        data


    /*********************************
    **          Variables           **
    *********************************/

    int channel;
    bool send_req_int;
    bool packet_just_sent;
    bool pending_request;
    int M_State;
    int S_State;

    /*********************************
    **        Constructor           **
    *********************************/

    SC_CTOR( ocp_interface) {

      SC_METHOD( core_interface );
       sensitive_pos << clk;

      SC_METHOD( ocp_master );
       sensitive_pos << clk;

      SC_METHOD( ocp_slave );
       sensitive_pos << clk;

    } // end constructor


    /*********************************
    **  SC_METHOD core_interface()  **
    ** Handles all communication    **
```

```
** with core.                    **
*********************************/

void core_interface() {

  if (nrst.read() == 0) {  //active reset
    core_hold.write(0);
    packet_just_sent = 0;
    packet_recv[DATA_WDTH] = 0;
    packet_send[M_ADDR_WDTH+DATA_WDTH] = 0;
  }

  else {

    if ( core_req.read() ) {
      core_hold.write(0);
      if (core_rw) {                                                    //
          receive
        channel = core_read.read();
        if ( (packet_recv[channel])[DATA_WDTH] ) {                      //
            active tag set, packet available
          core_data_out.write( (packet_recv[channel]).range(DATA_WDTH-1, 0) ); // pass
              data to core
          (packet_recv[channel])[DATA_WDTH] = 0;                        // tag
              packet as inactive
          packet_just_sent = 1;
        }
        else {                                                          // no
            data
          if (!packet_just_sent)
            core_hold.write(1);                                         // stall
                core
          packet_just_sent = 0;
        }
      }

      else {                                                            // send
        packet_just_sent = 0;
          if (packet_send[M_ADDR_WDTH+DATA_WDTH]) {                     //
              allready active packet in buffer
            core_hold.write(1);                                         // stall
                core
          }
          else {                                                        //
              buffer empty, store request
            packet_send[M_ADDR_WDTH+DATA_WDTH] = 1;                     // tag
                active
            packet_send.range(M_ADDR_WDTH+DATA_WDTH-1, DATA_WDTH) = core_write.read();
            packet_send.range(DATA_WDTH-1, 0) = core_data_in.read();
          }
      }
    }
    else {
      packet_just_sent = 0;
    }
  }
  return;
} // END OF core_interface


/*********************************
**    SC_METHOD ocp_slave()      **
```

```
** OCP slave, node is master.   **
*********************************/

void ocp_slave() {

  if (nrst.read() == 0) {                                          //active reset

  }

  else {

    if (SgetRequest(0)) {                                          // request from
        master, allways write
      if ( !(packet_recv[S_Address()])[DATA_WDTH] ) {              // buffer empty
          , store data
        (packet_recv[S_Address()])[DATA_WDTH] = 1;
        (packet_recv[S_Address()]).range(DATA_WDTH-1, 0) = SgetData();
        SRelease();
      }

    }

  }
  return;
} // END OF ocp_slave


/*********************************
**    SC_METHOD ocp_master()     **
** OCP master, node is slave.   **
*********************************/

void ocp_master() {

  if (nrst.read() == 0) {                                              //active
      reset

    M_State = 0;
    pending_request = 0;
  }

  else {

    switch(M_State) {

    case IDLE:
      if (packet_send[M_ADDR_WDTH+DATA_WDTH] == 1)  {                  // data
          waiting in buffer
        if(!MgetSBusy()) {                                             // slave
            not busy
          M_Address(packet_send.range(M_ADDR_WDTH+DATA_WDTH-1, DATA_WDTH));
          MputData(packet_send.range(DATA_WDTH-1, 0));
          MputWriteRequest();
          packet_send[M_ADDR_WDTH+DATA_WDTH] = 0;
          pending_request = 1;
          M_State = WAIT;
        }
      }
      break;

    case WAIT:
      if (MgetResponse(0))         // if slave response, no release necessary
```

```
            M_State = IDLE;
          break;

        }
        return;
      }
    } // END OF ocp_master


    /*************************
    ** OCP master functions  **
    *************************/

    // set target address
    void M_Address(sc_uint<M_ADDR_WDTH> Addr) {

      M_MAddr.write(Addr);
      return;

    } // END OF M_Address


    // set M_MData field
    void MputData(sc_uint<DATA_WDTH> Data) {

      sc_int<DATA_WDTH> temp;
      temp = Data;
      M_MData.write(temp);
      return;

    } // END OF MputData


    // check if slave is busy
    bool MgetSBusy() {

      return pending_request;

    } // END OF M_MgetSBusy


    // issue write request
    bool MputWriteRequest() {

      M_MCmd.write(WR);
      return pending_request;

    } // END OF M_MputWriteRequest


    // check if slave has responded
    bool MgetResponse(bool Release) {

      if (M_SCmdAccept.read()) {
        M_MCmd.write(IDLE);
        pending_request = 0;
        return 1;
      }
      else
        return 0;

    } // END OF MgetResponse
```

```
  /*************************
  **  OCP slave functions  **
  *************************/

  // return S_MAddr field
  int S_Address() {

    int i;
    i = S_MAddr.read();
    return i;

  } // END OF S_Address

  // return S_MData field
  sc_int<DATA_WDTH> SgetData() {

    return S_MData.read();

  }


  // test if master has sent request
  bool SgetRequest(bool Release) {

    S_SCmdAccept.write(0); // default value

    int temp;
    temp  = S_MCmd.read();
    if (temp == WR) {      // only write is active
      if (Release)
        S_SCmdAccept.write(1);
      return 1;
    }
    else
      return 0;

  } // END OF SgetRequest


  // acknowledge master request
  void SRelease() {

    S_SCmdAccept.write(1);
    return;

  }

}; // END OF SC_MODULE

#endif
```

A.23 output_core.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : output_core.cpp                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 25.12.2002                                   **
```

```
 **                                                                    **
 ** Function         : Mini-core simulation external data consumer.   **
 **                  : Stores received data in file.                  **
 ** Major revisions : 25.12.2002   Final version.                     **
 ************************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "declarations.h"

#define OUTPUT_LENGTH 150
#define CC 10

template<int CHANNEL, int DATA_WIDTH, int ADDR_WIDTH, const char *output_name>
SC_MODULE(output_core) {

  /*********************************
  **          Ports               **
  *********************************/

  sc_in_clk                          clk;
  sc_in< bool >                      nrst;        // active low, synchronous reset
  sc_out< bool >                     req;         // request to interface
  sc_out< bool >                     rw;          // send or receive (1=rec) to interface
  sc_in< bool >                      hold;        // hold from interface
  sc_out< sc_uint<DATA_WIDTH> >      data_out;    // data to interface
  sc_out< sc_uint<ADDR_WIDTH> >      write_addr;  // send address to interface
  sc_out< sc_uint<CHANNEL_WIDTH> >   read_addr;   // read channel to interface
  sc_in< sc_uint<DATA_WIDTH> >       data_in;     // data from interface

  int counter;
  int sr_count;
  double write_value;
  bool req_int;
  sc_fixed< DATA_WIDTH,SAMPLE_INT_WIDTH,SC_RND,SC_SAT > dataout_fx;
  sc_uint<DATA_WIDTH> output_temp;
  FILE *p_outfile;

  SC_CTOR( output_core ) {

    // open output file
    if ( (p_outfile = fopen(output_name, "w")) == NULL)  {
      printf("Error opening %s!!!\n", output_name);
      abort();
    }

    SC_METHOD( write );
    sensitive_pos << clk;

  } // constructor


  void write() {

     if (nrst.read() == 0) {
      req.write(0);
      counter = 0;
      sr_count = 0;
      req_int = 0;
    }
```

```
70        else {  // clockedge
            if (!hold.read()) {      // no hold from interface
              if (counter == CC) {
                if (sr_count < 3)
                  receive(0);
                else
                  counter = 0;
              }
              else {
                counter++;
80            }
            }
            else {
              if (req_int == 1)      // keep request high during hold
                req.write(1);
            }
          }
          return;
        }

90      void receive(int channel) {

          int value;  //debug

          if (sr_count == 0) {                    // 1st cycle of receive
            // send request to interface
            req.write(1);
            req_int = 1;
            rw.write(1);
            read_addr.write(channel);             // channel to read from
100         // make sure 2nd cycle is performed
            sr_count++;
          }

          else {                                  // 2nd cycle
            if (sr_count == 1) {
              sr_count++;
              req.write(0);
            }

110         else {                                // 3rd and last cycle of receive

              // store value from interface
              output_temp = data_in.read();

              // convert value to floating point
              int i;
              for (i=0; i<DATA_WIDTH; i++) {
                dataout_fx[i] = output_temp[i];
              }
120
              // convert to floating point
              write_value = dataout_fx;

              // write value to file
              fprintf(p_outfile, "%lf\n", write_value);
              if (fflush(p_outfile) == EOF)                    // make sure value is written
                  to file
                printf("Error writing to %s!!!\n", output_name);

              sr_count = 0;
```

```
130           req_int = 0;

              //debug
              value = data_in.read();
              printf("OUTPUT reading... \t value: %x\t from channel: %u \n", value, channel);
            }
          }
          return;
        }

140   }; // class
```

A.24 output_core_node

```
/*********************************************************************
 **                                                                 **
 ** Name             : output_core_node.cpp                        **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                     **
 ** Date             : 29.12.2002                                  **
 **                                                                 **
 ** Function         : Connects wrapped output core with generic OCP **
 **                  : node.                                       **
10 **                                                                 **
 ** Major revisions : 08.01.2003   First version.                   **
 **                                                                 **
 *********************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "ocp_generic_node.cpp"
20 #include "output_ocp_wrap.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *output_name, bool
    STAT_ON, const char *STAT_NAME>
SC_MODULE(output_core_node) {

  /*********************************
  **          Ports                **
  *********************************/

  sc_in_clk                          clk;
30  sc_in< bool >                      nrst;           // active low, synchronous reset
  sc_out< sc_uint<DATA_WIDTH> >      data_out;       // data to network
  sc_in< sc_uint<DATA_WIDTH> >       data_in;        // data from network
  sc_out< bool >                     req_out;        // request to network
  sc_in< bool >                      req_in;         // request from network
  sc_out< sc_uint<ADDR_WIDTH> >      addr_out;       // address to network
  sc_in< sc_uint<CHANNEL_WIDTH> >    addr_in;        // address (channel) from network
  sc_in< bool >                      hold_in;        // hold from network
  sc_out< bool >                     hold_out;       // hold to network

40

  /*********************************
  **          Signals              **
  *********************************/

  sc_signal< sc_uint<3> > nodeM_MCmd;               // OCP signals node master
  sc_signal< sc_uint<CHANNEL_WIDTH> > nodeM_MAddr; // ...
```

```
    sc_signal< sc_int<DATA_WIDTH> > nodeM_MData;      // ...
    sc_signal< bool > nodeM_SCmdAccept;               // ...
    sc_signal< sc_int<DATA_WIDTH> > nodeM_SData;      // ...
    sc_signal< sc_uint<2> > nodeM_SResp;              // ...
    sc_signal< sc_uint<3> > intM_MCmd;                // OCP signals interface master
    sc_signal< sc_uint<ADDR_WIDTH> > intM_MAddr;      // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_MData;       // ...
    sc_signal< bool > intM_SCmdAccept;                // ...
    sc_signal< sc_int<DATA_WIDTH> > intM_SData;       // ...
    sc_signal< sc_uint<2> > intM_SResp;               // ...


    /*********************************
    **          Components          **
    *********************************/

    output_ocp_wrap< ADDRESS, DATA_WIDTH, ADDR_WIDTH, output_name, STAT_ON, STAT_NAME > *
        output_wrap;
    ocp_generic_node< DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH > *node;


    /*********************************
    **         Constructor          **
    *********************************/

    SC_CTOR(output_core_node) {

      output_wrap = new output_ocp_wrap<ADDRESS, DATA_WIDTH, ADDR_WIDTH, output_name,
          STAT_ON, STAT_NAME > ("output_wrap");
      output_wrap->clk(clk);
      output_wrap->nrst(nrst);
      output_wrap->M_MCmd(intM_MCmd);
      output_wrap->M_MAddr(intM_MAddr);
      output_wrap->M_MData(intM_MData);
      output_wrap->M_SCmdAccept(intM_SCmdAccept);
      output_wrap->M_SData(intM_SData);
      output_wrap->M_SResp(intM_SResp);
      output_wrap->S_MCmd(nodeM_MCmd);
      output_wrap->S_MAddr(nodeM_MAddr);
      output_wrap->S_MData(nodeM_MData);
      output_wrap->S_SCmdAccept(nodeM_SCmdAccept);
      output_wrap->S_SData(nodeM_SData);
      output_wrap->S_SResp(nodeM_SResp);

      node = new ocp_generic_node<DATA_WIDTH, ADDR_WIDTH, CHANNEL_WIDTH> ("node");
      node->clk(clk);
      node->nrst(nrst);
      node->addr_in(addr_in);
      node->addr_out(addr_out);
      node->data_in(data_in);
      node->data_out(data_out);
      node->hold_out(hold_out);
      node->hold_in(hold_in);
      node->req_out(req_out);
      node->req_in(req_in);
      node->M_MCmd(nodeM_MCmd);
      node->M_MAddr(nodeM_MAddr);
      node->M_MData(nodeM_MData);
      node->M_SCmdAccept(nodeM_SCmdAccept);
      node->M_SData(nodeM_SData);
      node->M_SResp(nodeM_SResp);
      node->S_MCmd(intM_MCmd);
```

```
    node->S_MAddr(intM_MAddr);
    node->S_MData(intM_MData);
    node->S_SCmdAccept(intM_SCmdAccept);
    node->S_SData(intM_SData);
    node->S_SResp(intM_SResp);

  }
}; // END OF output_core_node
```

A.25 output_ocp_wrap.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : output_ocp_wrap.cpp                          **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 02.01.2002                                   **
 **                                                                 **
 ** Function         : Communications wrapper for output core.      **
 **                  : Connects core with OCP interface.            **
 **                                                                 **
 ** Major revisions : 02.01.2002   First version.                   **
 **                  : 14.01.2003   Statistics added.               **
 **                                                                 **
 *********************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "output_core.cpp"
#include "ocp_interface.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *output_name, bool
    STAT_ON, const char *STAT_NAME >
SC_MODULE(output_ocp_wrap) {


  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<3> >              M_MCmd;          // OCP signals as master
  sc_out< sc_uint<ADDR_WIDTH> >     M_MAddr;         // ...
  sc_out< sc_int<DATA_WIDTH> >      M_MData;         // ...
  sc_in< bool >                     M_SCmdAccept;    // ...
  sc_in< sc_int<DATA_WIDTH> >       M_SData;         // ...
  sc_in< sc_uint<2> >               M_SResp;         // ...
  sc_in< sc_uint<3> >               S_MCmd;          // OCP signals as slave
  sc_in< sc_uint<CHANNEL_WIDTH> >   S_MAddr;         // ...
  sc_in< sc_int<DATA_WIDTH> >       S_MData;         // ...
  sc_out< bool >                    S_SCmdAccept;    // ...
  sc_out< sc_int<DATA_WIDTH> >      S_SData;         // ...
  sc_out< sc_uint<2> >              S_SResp;         // ...


  /*************
   ** Signals **
   *************/
```

```
50
    sc_signal< bool >                       core_rw;          // read/write from core
    sc_signal< sc_uint<DATA_WIDTH> >        core_data_out;    // data from core
    sc_signal< sc_uint<ADDR_WIDTH> >        core_write;       // send address from core
    sc_signal< sc_uint<CHANNEL_WIDTH> >     core_read;        // read address from core
    sc_signal< sc_uint<DATA_WIDTH> >        core_data_in;     // data to core
    sc_signal< bool >                       core_hold;        // hold to core
    sc_signal< bool >                       core_req;         // request from core


60  /****************
     ** Variables  **
     ****************/

    int recv_req_count;
    int send_req_count;
    int hold_count;
    int cc_count;
    FILE *p_stat;

70
    /****************
     ** Components **
     ****************/

    output_core<0, DATA_WIDTH, ADDR_WIDTH, output_name > *output;
    ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > *interface;


    SC_CTOR(output_ocp_wrap) {
80
      // Output core
      output = new output_core< 0, DATA_WIDTH, ADDR_WIDTH, output_name > ("output");

      output->clk(clk);
      output->nrst(nrst);
      output->req(core_req);
      output->rw(core_rw);
      output->hold(core_hold);
      output->data_out(core_data_in);
90    output->write_addr(core_write);
      output->read_addr(core_read);
      output->data_in(core_data_out);


      // Interface
      interface = new ocp_interface< ADDRESS, DATA_WIDTH, CHANNEL_WIDTH, ADDR_WIDTH > ("
          Interface");

      interface->clk(clk);
      interface->nrst(nrst);
100   interface->core_req(core_req);
      interface->core_rw(core_rw);
      interface->core_data_out(core_data_out);
      interface->core_write(core_write);
      interface->core_read(core_read);
      interface->core_data_in(core_data_in);
      interface->core_hold(core_hold);
      interface->M_MCmd(M_MCmd);
      interface->M_MAddr(M_MAddr);
      interface->M_MData(M_MData);
110   interface->M_SCmdAccept(M_SCmdAccept);
```

```
    interface->M_SData(M_SData);
    interface->M_SResp(M_SResp);
    interface->S_MCmd(S_MCmd);
    interface->S_MAddr(S_MAddr);
    interface->S_MData(S_MData);
    interface->S_SCmdAccept(S_SCmdAccept);
    interface->S_SData(S_SData);
    interface->S_SResp(S_SResp);

    // statistics generating
    SC_METHOD( statistics ); // Creates run-time report and statistics
    sensitive_neg << clk;

  }


   /*******************************
    ** statistics()               **
    ** Outputs hold and req count **
    **  to file.                  **
    *******************************/
  void statistics() {

    int i;

    if (nrst == 0) {
      recv_req_count = 0;
      send_req_count = 0;
      hold_count = 0;
      cc_count = 0;
    }
    else {
      if (STAT_ON) {

        // create statistics
        if ( (p_stat = fopen(STAT_NAME, "w")) == NULL)  {
          printf("Error creating statistics file!!!\n");
          abort();
        }

        if (core_req) {
          if (core_rw)
            recv_req_count++;
          else
            send_req_count++;
        }
        if (core_hold)
          hold_count++;

        fprintf(p_stat,"%u\n%u\n%u\n",recv_req_count, send_req_count, hold_count);

        fclose(p_stat);
      }

      cc_count++;

    }
  } // END OF statistics

};
```

A.26 output_wrap.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : output_wrap.cpp                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 25.12.2002                                   **
 **                                                                 **
 ** Function         : Communications wrapper for output core.      **
 **                  : Connects core with interface.                **
 **                                                                 **
 ** Major revisions : 25.12.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/


#define SC_INCLUDE_FX

#include "systemc.h"
#include "output_core.cpp"
#include "simple_interface.cpp"

template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH, const char *output_name >
SC_MODULE(output_wrap) {

  /***********
   ** Ports **
   ***********/

  sc_in<bool> clk;
  sc_in<bool> nrst;
  sc_out< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > net_send_packet;   // packet to network
  sc_in< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > net_recv_packet; // packet from network
  sc_in< bool >                    send_hold;                   // send hold from network
  sc_out< bool >                   recv_hold;                   // receive hold to
      network
  sc_out< bool >                   send_req;                    // send request to
      network
  sc_in< bool >                    recv_req;                    // receive request from
      network

  /*************
   ** Signals **
   *************/

  sc_signal< bool >                    core_rw;          // read/write from core
  sc_signal< sc_uint<DATA_WIDTH> >     core_data_out;    // data from core
  sc_signal< sc_uint<ADDR_WIDTH> >     core_write;       // send address from core
  sc_signal< sc_uint<CHANNEL_WIDTH> >  core_read;        // read address from core
  sc_signal< sc_uint<DATA_WIDTH> >     core_data_in;     // data to core
  sc_signal< bool >                    core_hold;        // hold to core
  sc_signal< bool >                    core_req;         // request from core

  /***************
   ** Components **
   ***************/

  output_core<0, DATA_WIDTH, ADDR_WIDTH, output_name > *output;
  simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > *interface;

  SC_CTOR(output_wrap) {
```

```
      // Output core
      output = new output_core< 0, DATA_WIDTH, ADDR_WIDTH, output_name > ("output");

      output->clk(clk);
      output->nrst(nrst);
      output->req(core_req);
      output->rw(core_rw);
      output->hold(core_hold);
      output->data_out(core_data_in);
      output->write_addr(core_write);
      output->read_addr(core_read);
      output->data_in(core_data_out);


      // Interface
      interface = new simple_interface< ADDRESS, DATA_WIDTH, ADDR_WIDTH > ("Interface");

      interface->clk(clk);
      interface->nrst(nrst);
      interface->core_req(core_req);
      interface->core_rw(core_rw);
      interface->core_data_out(core_data_out);
      interface->core_write(core_write);
      interface->core_read(core_read);
      interface->core_data_in(core_data_in);
      interface->core_hold(core_hold);
      interface->net_send_packet(net_send_packet);
      interface->net_recv_packet(net_recv_packet);
      interface->send_hold(send_hold);
      interface->recv_hold(recv_hold);
      interface->send_req(send_req);
      interface->recv_req(recv_req);

    }
  };
```

A.27 simple_interface.cpp

```
  /*********************************************************************
   **                                                                 **
   ** Name             : simple_interface.cpp                         **
   **                                                                 **
   ** Author           : Karsten Larsen, c971833                      **
   ** Date             : 29.11.2002                                   **
   **                                                                 **
   ** Function         : Communications interface for the mini-cores. **
   ** Major revisions : 21.10.2002   First version.                   **
   **                  : 29.11.2002   Support for 4 channels per core **
   **                  :              added.                           **
   **                                                                 **
   *********************************************************************/

  #ifndef _simple_interface_cpp
  #define _simple_interface_cpp

  #include "systemc.h"

  template< int ADDRESS, int DATA_WIDTH, int ADDR_WIDTH >
  SC_MODULE(simple_interface) {
```

```
/*********************************
**          Ports                **
*********************************/

sc_in_clk                          clk;
sc_in< bool >                      nrst;                        // active low,
    synchronous reset
sc_in< bool >                      core_req;                    // request from core
sc_in< bool >                      core_rw;                     // read/write from core
sc_in< sc_uint<CHANNEL_WIDTH> >    core_read;                   // receive address from
    core
sc_out< sc_uint<DATA_WIDTH> >      core_data_out;               // data from core
sc_in< sc_uint<ADDR_WIDTH> >       core_write;                  // send address from core
sc_in< sc_uint<DATA_WIDTH> >       core_data_in;                // data to core
sc_out< bool >                     core_hold;                   // hold to core
sc_out< sc_uint<ADDR_WIDTH+DATA_WIDTH+1> > net_send_packet;     // packet to network
sc_in< sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> > net_recv_packet;   // packet from network
sc_in< bool >                      send_hold;                   // send hold from network
sc_out< bool >                     recv_hold;                   // receive hold to
    network
sc_out< bool >                     send_req;                    // send request to
    network
sc_in< bool >                      recv_req;                    // receive request from
    network

/*********************************
**          Registers            **
*********************************/

sc_uint<CHANNEL_WIDTH+DATA_WIDTH+1> packet_recv[CHANNEL_COUNT];  //buffered receive
    packets, N channels, tag(1):addr:data
sc_uint<ADDR_WIDTH+DATA_WIDTH+1> packet_send;                    //buffered send packet
    , tag(1):addr:data
int channel;
bool recv_hold_int;
bool send_hold_int;
bool send_req_int;
bool packet_just_sent;

/*********************************
**        Constructor            **
*********************************/

SC_CTOR( simple_interface) {

  SC_METHOD( interface_read );
  sensitive_pos << clk;

  SC_METHOD( interface_write );
  sensitive_neg << clk;

} // end constructor

/*********************************
**       interface_read()        **
** Action performed on rising    **
** clockedge.                    **
*********************************/

void interface_read() {

  if (nrst.read() == 0) {                    //active reset
```

```
    core_hold.write(0);
    recv_hold.write(0);
    packet_just_sent = 0;
    packet_recv[CHANNEL_WIDTH+DATA_WIDTH] = 0;
    packet_send[ADDR_WIDTH+DATA_WIDTH] = 0;
  }

  else {

    /*******************************
     **  send or receive from core  **
     *******************************/

    // recv_data_ready = 0;
    if ( core_req.read() ) {
      core_hold.write(0);
      if (core_rw) { // receive
        channel = core_read.read();
        if ( (packet_recv[channel])[CHANNEL_WIDTH+DATA_WIDTH] ) {              // active
            tag set, packet available
          core_data_out.write( packet_recv[channel].range(DATA_WIDTH-1, 0) ); // pass
              data to core
          (packet_recv[channel])[CHANNEL_WIDTH+DATA_WIDTH] = 0;                // tag
              packet as inactive
          packet_just_sent = 1;
          recv_hold_int = 0;
        }
        else {                                          // no data
          if (!packet_just_sent)
            core_hold.write(1);                         // stall core
          packet_just_sent = 0;
        }
      }

      else {          // send
        packet_just_sent = 0;
        if (send_hold.read()) {
          core_hold.write(1);
        }
        else {
          if (packet_send[ADDR_WIDTH+DATA_WIDTH]) { // allready active packet in buffer
            core_hold.write(1);                     // stall core
          }
          else {                                    // buffer empty, store request
            packet_send[ADDR_WIDTH+DATA_WIDTH] = 1; // tag active
            packet_send.range(ADDR_WIDTH+DATA_WIDTH-1, DATA_WIDTH) = core_write.read();
            packet_send.range(DATA_WIDTH-1, 0) = core_data_in.read();
          }
        }
      }
    }
    else
      packet_just_sent = 0;

    /*******************************
     **     receive from network     **
     *******************************/

    if ( recv_req.read() ) {                                    // packet from network
        waiting
      channel = (net_recv_packet.read()).range(CHANNEL_WIDTH+DATA_WIDTH-1, DATA_WIDTH);
      if ( (packet_recv[channel])[CHANNEL_WIDTH+DATA_WIDTH] ) { // buffer full
```

```
            recv_hold_int = 1;                                          // stall network on
                falling clk edge
          }
          else {                                                        // store in buffer
            recv_hold_int = 0;
140         packet_recv[channel] = net_recv_packet.read();
            (packet_recv[channel])[CHANNEL_WIDTH+DATA_WIDTH] = '1'; // tag as new data
          }
        }

        /*********************************
         **         send to network       **
         *********************************/

        if (!send_req_int) {                                            // if no send in
            progress
150       if (packet_send[ADDR_WIDTH+DATA_WIDTH]) {                     // packet in send
              buffer
            send_req_int = 1;
          }
        }
      }
      return;
    } // END OF interface_read

    /*********************************
    **       interface_write()       **
160 ** Writing on falling clockedge **
    *********************************/

      void interface_write() {

        if (nrst.read() == 0) {  // reset
          recv_hold.write(0);
          send_req.write(0);
        }

170     else {
          // network write
          recv_hold.write(recv_hold_int);

          send_req.write(send_req_int);
          if (send_req_int) {
            net_send_packet.write(packet_send);
            if (!send_hold.read()) {
              send_req_int = 0;
              packet_send[ADDR_WIDTH+DATA_WIDTH] = 0; // tag buffer as empty
180         }
          }
        }
        return;
      } // END OF interface_write


};  // END OF SC_MODULE

#endif
```

A.28 stimuli.cpp

```
/*****************************************************************************
```

```
/*************************************************************************
** Name             : stimuli.cpp                                       **
**                                                                      **
** Author           : Karsten Larsen, c971833                           **
** Date             : 14.10.2002                                        **
**                                                                      **
** Function         : Supplies active low reset for mini-core system.   **
**                                                                      **
** Major revisions : 14.10.2002    Final version.                       **
**                                                                      **
*************************************************************************/


#include "systemc.h"
#include "declarations.h"

SC_MODULE(stimuli) {

  sc_in_clk        clk;
  sc_out< bool >   nrst;

  //Constructor
  SC_CTOR( stimuli ) {
    SC_THREAD( stim_gen );

    sensitive_pos ( clk );  // synchronous
  }

  void stim_gen()
  {

    int i = 0;

    while(true) {
      if (i < 2)
        nrst.write(0);
      else
        nrst.write(1);

      i++;

      wait();
    }
  }

};
```

Appendix B

Source Code for Programming Tools

This appendix contains source code of the programs developed to assist is programming of the FIR and IIR processors.

The programs are:

- *coeff* - converts coefficients from the format employed in the assembler output to the format employed in the behavioral SystemC model.
- *postasm_fir* - converts the binary output of the assembler to the instruction format of the FIR mini-core.
- *postasm_fir* - converts the binary output of the assembler to the instruction format of the IIR mini-core.

B.1 coeff.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : coeff_conv.cpp                               **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 28.12.2002                                   **
 **                                                                 **
 ** Function         : Converts array of integers to hex and writes **
 **                  : them to file.                                **
10 **                  : Used to convert coefficients from VHDL design. **
 ** Major revisions : 28.12.2002   Final version.                  **
 **                                                                 **
 *********************************************************************/

#define SC_INCLUDE_FX

#include "systemc.h"

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

```
20
    int cmem[100];
    int i, N, memword;
    FILE *p_cfile;

    // input coefficient file
    if ( (p_cfile = fopen("cin.bin", "r")) == NULL)  {
      printf("Error opening input file !!!\n");
      abort();
    }
30
    // read integers
    i = 0;
    while (fscanf(p_cfile, "%u", &memword) != EOF) {
      cmem[i] = memword;
      i++;
    }
    N = i;              // words read
    fclose(p_cfile);

40  // open output file
    if ( (p_cfile = fopen("cmem.bin", "w")) == NULL)  {
      printf("Error creating output file !!!\n");
      abort();
    }

    // write to file
    for (i=0; i<N; i++) {
      fprintf(p_cfile,"0x%x\n", cmem[i]);
    }
50
    fclose(p_cfile);

    return(0);
}
```

B.2 postasm_fir.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : postasm_fir.cpp                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 14.11.2002                                   **
 **                                                                 **
 ** Function         : Postprocesses assembler output to change     **
 **                  : format into standard FIR processor format.   **
10 ** Major revisions : 14.11.2002   Final version.                 **
 **                                                                 **
 *********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define IMM_WIDTH 8

20 void main(int argc, char *argv[]) {

    FILE *asmfile;
    FILE *binfile;
```

```
    char bin_to_hex(char []);
    char memstring1[17];
    char memstring2[17];
    char thrash1[17];
    char thrash2[17];
30  char op_flag[9] = "00000000";
    char immediate[IMM_WIDTH+1];
    char instruction_bin[9+IMM_WIDTH];
    char instruction_hex[5+IMM_WIDTH/4];
    char hex_prefix[3];
    char temp[11];
    char nibble[5];

    int i;

40  //input file
    if ( (asmfile = fopen(argv[1], "r")) == NULL)  {
      printf("Error opening input file: %s\n", argv[1]);
      abort();
    }

    //output file
    binfile = fopen(argv[2], "w");

    //read bits from input file
50  while (fgets(thrash1, 18, asmfile) != NULL) {
      fgets(memstring1, 18, asmfile);
      memstring1[16] = '\0';
      fgets(thrash2, 18, asmfile);
      fgets(memstring2, 18, asmfile);
      memstring2[16] = '\0';

      // slice relevant bits
      for (i=0; i<IMM_WIDTH; ++i)
        immediate[i] = memstring1[i+16-IMM_WIDTH];
60    immediate[IMM_WIDTH] = '\0';

      for (i=0; i<8; i++)
        op_flag[i] = memstring2[i+8];
      op_flag[IMM_WIDTH] = '\0';

      instruction_hex[0] = '0';
      instruction_hex[1] = 'x';
      instruction_hex[4+IMM_WIDTH/4] = '\n';

70    // split into 4 bit nibbles and convert to hex
      nibble[4] = '\0';

      for (i=0; i<4; i++)
        nibble[i] = op_flag[i];                    // opcode
      instruction_hex[2] = bin_to_hex(nibble);

      for (i=0; i<4; i++)
        nibble[i] = op_flag[i+4];                  // flags
      instruction_hex[3] = bin_to_hex(nibble);
80
      int j;
      for(j=0; j<IMM_WIDTH/4; j++) {
        for (i=0; i<4; i++)
          nibble[i] = immediate[j*4+i];            // flags
        instruction_hex[4+j] = bin_to_hex(nibble);
```

```
      }
      for (i=0; i<=4+IMM_WIDTH/4; i++)
        fputc(instruction_hex[i], binfile);          // write to output file

    } //while

    fclose(binfile);
    fclose(asmfile);

    printf("Output written to: %s\n", argv[2]);

}//main


// converts 4 bit nibble into hex char
int bin_to_hex(char a[]) {

  int res = 0;
  char hex_res;

  if (a[0] == '1')
    res += 8;

  if (a[1] == '1')
    res += 4;

  if (a[2] == '1')
    res += 2;

  if (a[3] == '1')
    res += 1;

  if (res >= 0 && res <= 9)
    hex_res = res+48;        //convert to char ('0' = 30)
  else
    hex_res = res+97-10;     // convert to char ('a' = 97)

  return hex_res;
}
```

B.3 postasm_iir.cpp

```
/*********************************************************************
 **                                                                 **
 ** Name             : postasm_iir.cpp                              **
 **                                                                 **
 ** Author           : Karsten Larsen, c971833                      **
 ** Date             : 02.12.2002                                   **
 **                                                                 **
 ** Function         : Postprocesses assembler output to change     **
 **                  : format into standard IIR processor format.   **
 ** Major revisions : 02.12.2002   Final version.                   **
 **                                                                 **
 *********************************************************************/


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
```

```
20
    FILE *asmfile;
    FILE *binfile;

    char bin_to_hex(char []);
    char memstring1[17];
    char memstring2[17];
    char thrash1[17];
    char thrash2[17];
    char opcode[6];// = "0000";
30  char immediate[9];// = "0000000";
    char instruction_bin[17];
    char instruction_hex[5];
    char temp[14];
    char nibble[7];
    char zeros[4] = "000";

    int i;

    //input file
40  if ( (asmfile = fopen(argv[1], "r")) == NULL)   {
      printf("Error opening input file: %s\n", argv[1]);
      abort();
    }

    //output file
    if ( (binfile = fopen(argv[2], "w"))== NULL) {
      printf("Error creating output file: %s\n", argv[2]);
      abort();
    }
50
    //read bits from input file
    while (fgets(thrash1, 18, asmfile) != NULL) {
      fgets(memstring1, 18, asmfile);
      memstring1[16] = '\0';
      fgets(thrash2, 18, asmfile);
      fgets(memstring2, 18, asmfile);
      memstring2[16] = '\0';


60    // slice relevant bits
      for (i=0; i<8;++i)
        immediate[i] = memstring1[i+8];
      immediate[8] = '\0';

      for (i=0; i<5; i++)
        opcode[i] = memstring2[i+11];
      opcode[5] = '\0';

      for (i=0; i<8; i++)
70      instruction_bin[i] = immediate[7-i];
      for (i=0; i<5; i++)
        instruction_bin[i+8] = opcode[4-i];
      for (i=0; i<3; i++)
        instruction_bin[i+13] = '0';
      instruction_bin[16] = '\0';

      for (i=0; i<16; i++)
        printf("%c", instruction_bin[i]);
      printf("\n");
80
      // construct hex header
```

```
    instruction_hex[0] = '0';
    instruction_hex[1] = 'x';

    // split into 4 bit nibbles and convert to hex
    for (i=0; i<4; i++)
      nibble[3-i] = instruction_bin[i+12];
    instruction_hex[2] = bin_to_hex(nibble);

    for (i=0; i<4; i++)
      nibble[3-i] = instruction_bin[i+8];
    instruction_hex[3] = bin_to_hex(nibble);

    for (i=0; i<4; i++)
      nibble[3-i] = instruction_bin[i+4];
    instruction_hex[4] = bin_to_hex(nibble);

    for (i=0; i<4; i++)
      nibble[3-i] = instruction_bin[i];
    instruction_hex[5] = bin_to_hex(nibble);

    instruction_hex[6] = '\n';

    for (i=0; i<7; i++)
      fputc(instruction_hex[i], binfile);  // write to output file

  } //while

  fclose(binfile);
  fclose(asmfile);

  printf("Output written to: %s\n", argv[2]);

  return 0;
}//main


// converts 4 bit nibble into hex char
int bin_to_hex(char a[]) {

  int res = 0;
  char hex_res;

  if (a[0] == '1')
    res += 8;

  if (a[1] == '1')
    res += 4;

  if (a[2] == '1')
    res += 2;

  if (a[3] == '1')
    res += 1;

  if (res >= 0 && res <= 9)
    hex_res = res+48;         //convert to char ('0' = 30)
  else
    hex_res = res+97-10;      // convert to char ('a' = 97)

  return hex_res;
}
```

Appendix C

Filterbank Algorithm Source Code

This appendix contains the source code of the assembler program *filterbank* used to test the mini-core system as implemented in this thesis. *filterbank_fir1* and *filterbank_fir2* [7], [14] separate the input signal into 7 frequency bands. *filterbank_iir1-3* add the frequency bands together and transmit the result to the OUTPUT module. *filterbank_fir3* keeps FIR 3 idle.

To execute *filterbank* on the behavioral SystemC platform, the data segments of *filterbank_fir1* and *filterbank_fir2* must be removed. The coefficient data segment (cm) must be processed through the *coeff* program and the output placed in a separate file. The filter data segment (fm) must also be placed in a separate file. Notice that the assembler code employs integer format, while the filter file of the behavioral SystemC platform employs hexadecimal format. Conversion must be performed manually.

C.1 filterbank_fir1.asm

```
#####################################################
#                                                   #
# Author   : Niels Haandbaek                        #
#          : Comments Updated by Karsten Larsen     #
# Name     : filterbank_fir1                        #
# Purpose  : Half of the filterbank from Ozguns     #
#          : thesis.                                #
#                                                   #
#####################################################

#####################################################
# Memory setup:                                     #
#####################################################

.data(cm, addr = 0x100)
  63937
  9473
  16384
  63937
```

```
20    9473
      16384
      64353
      9345
      16384
      63232
      64641
      57344
      4863
      16384
30    8192
      16384

    .data(fm, addr = 0x300)
    # ln bcp cp bdp dp1 dp2
      49  0 0   0 0 48
      35  3 0  49 0 34
      13  6 0  84 0 12
      15  9 0  97 0 14
       3 14 0 112 0  2
40     2  0 0 115 0  1

    .code(addr = 0x000, imm_width = 8)

    ##################################################
    # The program:                                   #
    ##################################################

    start:
    receive tmp1, 0          # Get input sample from channel 0.
50
    ##################################################
    # Filter H1: (LN = 49)                           #
    ##################################################
    switch clr,0             # Select first filter.
    mov    regdm,tmp1        # Move sample to delay-line.
    asmacc clr,16
    asmacc 8
    macc   fin,comp, 24

60  ##################################################
    # Filter H2: (LN = 35)                           #
    ##################################################
    switch clr, 1            # Select filter.
    send   comp,4            # Send complementary output to FIR1, 0
    mov    regdm, macc       # Move previous results in delay-line.
    addi   mod, dp1,10       # Move pointer forward in delay-line.
    asmacc clr,8
    asmacc 4
    macc   fin,comp,12
70  subi   mod, dp2, 10      # Extra adjustment of dp2 to account for
                             # delay-line with irregular structure.

    ##################################################
    # Filter H4:                                     #
    ##################################################
    switch clr, 2            # Select filter.
    store  comp,dm,117       # Store result from previous computation in temporary
                             # variable.
    mov    regdm,macc        # Move sample to delay-line.
80  asmacc clr, 4
    asmacc 2
```

```
macc    fin,comp,6

#####################################################
# Filter H8:                                        #
#####################################################
switch  clr, 4          # Select filter.
mov     regdm, macc     # Move sample to delay-line.
asmacc  clr,1
macc    fin,1

#####################################################
# Filter H8d:                                       #
#####################################################
switch  clr, 5          # Select filter.
send    macc,12         # Send B1 to IIR1 channel 0
mov     regdm,comp      # Move sample to delay-line.
addi    mod,dp1,1       # Move pointer.
mov     dmreg,tmp3      # Get value in delay-line (tab coef is 1.0)
nop
nop
send    tmp3,13         # Send B6 (IIR1 channel 2)

#####################################################
# Filter H5:                                        #
#####################################################
load    tmp1,dm, 117    # Get the output from filter H4.
switch  clr, 3          # Select filter.
mov     regdm,tmp1      # Move sample to delay-line.
asmacc  clr,2
asmacc  2
asmacc  2
asmacc  1
macc    fin,7
nop
send    macc,14         # Send B4 (IIR1 channel 3)

jmp start
```

C.2 filterbank_fir2.asm

```
#####################################################
#                                                   #
# Author  : Niels Haandbaek                         #
#         : Comments updated by Karsten Larsen      #
# Name    : filterbank_fir2                         #
# Purpose : Half of the filterbank from Ozguns      #
#         : thesis.                                 #
#                                                   #
#####################################################

#####################################################
# Memory setup:                                     #
#####################################################

.data(cm, addr = 0x500)
  64769
  832
  62207
  9473
  16384
  64193
```

```
    9473
    16384
    256
    63488
    9729
    16384
    62207
    8287
    16384

.data(fm, addr = 0x700)
# ln bcp cp bdp dp1 dp2
  29   0  0   0   0  28
   7   5  0  29   0   6
  31   8  0  36   0  30
  25  12  0  67   0  24

.code(addr = 0x000, imm_width = 8)

#####################################################
# The program:                                      #
#####################################################

start:
receive tmp1, 0          # Get input sample from channel 0.
#lset   tmp1, 1

#####################################################
# Filter H3:                                        #
#####################################################
switch clr, 0            # Select filter.
mov    regdm, tmp1       # Move sample to delay-line.
asmacc clr, 4
asmacc 4
asmacc 4
asmacc 2
macc   fin,comp,14

#####################################################
# Filter H6:                                        #
#####################################################
switch clr, 1            # Select H6
store  comp, dm, 92      # Store output from H3 in temporary variable.
mov    regdm, macc       # Put result in delay-line.
asmacc clr, 2
asmacc 1
macc   fin, 3

#####################################################
# Filter H10:                                       #
#####################################################
switch clr, 3            # Select the filter.
mov    regdm, macc       # Move the result to the delay-line.
asmacc clr, 8
asmacc 4
macc   fin, comp, 12
nop
nop
nop

#####################################################
# Filter H7:                                        #
```

```
    ##################################################
    switch clr,2            # Select filter.
    send   macc, 16         # Send B2 (IIR2 channel 0)
    nop
    nop
    nop
 90 load   dm, tmp1, 92     # Get the output from H3.
    send   comp, 17         # Send B3 (IIR2 channel 1)
    mov    regdm, tmp1      # Put sample in delay-line.
    asmacc clr, 6
    asmacc 6
    asmacc 3
    macc   fin,comp,15
    nop                     # delay-slot of one for the macc register.
    send   macc, 18         # Send B7 (IIR2 channel 2)
    nop
100 nop
    nop
    nop
    nop
    send   comp, 19         # Send B5 (IIR2 channel 3)

    jmp start
```

C.3 filterbank_fir3.asm

```
    ##################################################
    #                                                #
    # Author  : Karsten Larsen                       #
    # Name    : filterbank_fir3                      #
    # Purpose : Keeps FIR3 idle                      #
    #                                                #
    ##################################################

    .code(addr = 0x000, imm_width = 8)
 10
    ##################################################
    # The program:                                   #
    ##################################################

    start:
            nop
            jmp     start
```

C.4 filterbank_iir1.asm

```
    ##################################################
    #                                                #
    # Author  : Karsten Larsen                       #
    # Name    : filterbank_iir1                      #
    # Purpose : Adds data from 3 input channels      #
    #                                                #
    ##################################################

    .code (addr=0x000,reg_width=2, read_width=2, write_width=6, biquad_width=1,imm_width=3,
        jump_width=5)
 10
    start:
            receive r0, 0
            receive r1, 1
```

```
        add     r0, r1
        receive r1, 2
        add     r0, r1
        send    r0, 20   #send to iir3(0)
        jump start
        nop
```

C.5 filterbank_iir2.asm

```
#####################################################
#                                                   #
# Author  : Karsten Larsen                          #
# Name    : filterbank_iir2                         #
# Purpose : Adds data from all 4 input channels     #
#                                                   #
#####################################################

.code (addr=0x000,reg_width=2, read_width=2, write_width=6, biquad_width=1,imm_width=3,
    jump_width=5)

start:
        receive r0, 0
        receive r1, 1
        add     r0, r1
        receive r1, 2
        add     r0, r1
        receive r1, 3
        add     r0, 1
        send    r0, 21   #send to iir1(1)
        jump start
        nop
```

C.6 filterbank_iir3.asm

```
#####################################################
#                                                   #
# Author  : Karsten Larsen                          #
# Name    : filterbank_iir3                         #
# Purpose : Adds data from 2 input channels         #
#                                                   #
#####################################################


.code (addr=0x000,reg_width=2, read_width=2, write_width=6, biquad_width=1,imm_width=3,
    jump_width=5)

start:
        receive r0, 0
        receive r1, 1
        add     r0, r1
        send    r0, 24  #IO output
        jump start
        nop
```