TAMPERE UNIVERSITY OF TECHNOLOGY

OTTO ESKO

ASIP INTEGRATION AND VERIFICATION FLOW FOR FPGA

Master of Science Thesis

Examiners: Prof. Jarmo Takala and Mr. Pekka Jääskeläinen, M.Sc.
Examiners and topic approved in the Computing and Electrical Engineering Faculty Council meeting on March 3 2010

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
ESKO, OTTO OLAVI: ASIP Integration and Verification Flow for FPGA
Master of Science Thesis, 54 pages
June 2011
Major: Embedded systems
Examiner: Prof. Jarmo Takala and Mr. Pekka Jääskeläinen, M.Sc.
Keywords: application-specific instruction-set processors, FPGA, integration, TTA

Over the years, user-programmable logic devices, such as FPGAs, have become a popular platform for testing and implementing hardware designs. *Intellectual Property (IP)* components and synthesizable processor cores allow complex design to be implemented in a reasonable time, thanks to design reuse and the flexibility provided by programmability. Unfortunately, the performance of *General Purpose Processors (GPP)* is often inadequate, and creating custom fixed function hardware implementations to boost the performance is often too time consuming and expensive.

Application-Specific Instruction-set Processors (ASIP) are one solution to match the performance of a custom fixed function hardware design and the flexibility of software by tailoring the processor architecture to match the specific application. In order to decrease the design time, and, thus, increase the productivity, a practical processor design environment is needed. *TTA-based Co-design Environment (TCE)*, developed at Tampere University of Technology, allows the designer to tailor ASIPs based on the *Transport Triggered Architecture (TTA)* processor template and to generate ASIP implementations. However, previously the TTA ASIPs had to be manually integrated into the target platform, which restrains the otherwise fluent design flow.

For this thesis, an automatic integration framework called Platform Integrator was created for TCE. The purpose of the framework is to automate the integration flow of TTA ASIPs to various FPGA platforms in order to reduce the design time. The design, implementation and verification of the Platform Integrator framework and three distinct Platform Integrator implementations are described in the thesis.

Another part of this thesis documents the verification flow of TTA ASIPs. The thesis introduces a new verification tool called TTA Unit Tester which is designed and implemented to complete the verification flow. The purpose of the TTA Unit Tester is to automatically verify the processor datapath resources. The different steps of the verification flow are utilized to verify the ASIP implementations created with the Platform Integrators.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
ESKO, OTTO OLAVI: ASIP Integration and Verification Flow for FPGA
Diplomityö, 54 sivua
Kesäkuu 2011
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Jarmo Takala ja DI Pekka Jääskeläinen
Avainsanat: sovelluskohtaiset prosessorit, FPGA, integrointi, TTA

Uudelleenohjelmoitavien logiikkapiirien, kuten FPGA-piirien, käyttö on vuosien saatossa yleistynyt digitaalisen logiikan toteuttamisalustana. FPGA-piirille syntesoitavat prosessoriytimet mahdollistavat toteutuksen ohjelmoitavuuden, mikä lisää suunnittelun juostavuutta. Lisäksi uudelleenkäytettävät loogiikkakomponentit, eli niin sanotut IP-lohkot, helpottavat ja nopeuttavat suunnittelua. Näiden avulla monimutkaisiakin järjestelmiä voidaan toteuttaa kohtuullisessa ajassa. Yleiskäyttöisten prosessorien suorituskyky ei kuitenkaan ole aina riittävä, eikä sovelluskohtaisen logiikkakomponentin tekeminen ole aina mahdollista liian suurten toteutuskustannuksien vuoksi. Tällöin ratkaisua puuttuvaan suorituskykyyn pitää etsiä muualta.

Sovelluskohtaiset prosessorit (Application-Specific Instruction-set Processor, ASIP) ovat yksi tapa yhdistää logiikkatoteutuksen suorituskyky ja ohjelmoitavuuden tarjoamat edut. Yleiskäyttöistä prosessoria parempi suorituskyky saavutetaan räätälöimällä prosessori sopimaan mahdollisimman hyvin tietyn sovelluksen suorittamiseen. Jotta ASIP:n toteutusaika ja -kustannukset pysyisivät kurissa, tarvitaan toimiva ja helppokäyttöinen prosessorisuunnitteluohjelmisto. Tampereen Teknillisessä Yliopistossa kehitettävä *TTA-based Co-design Environment*-työkaluympäristö (TCE) mahdollistaa *Transport Triggered Architecture*-prosessoriarkkitehtuuriin (TTA) perustuvien prosessoreiden räätälöimisen ja toteuttamisen. Aiemmin sovelluskohtaisen TTA-prosessorin integroiminen halutulle kohdealustalle, kuten FPGA-piirille, vaati kuitenkin käsityötä, joka hidasti muutoin sujuvaa suunnitteluprosessia.

Diplomityössä suunniteltiin ja toteutettiin ohjelmistokehys niin sanotulle alustaintegraattorille (engl. Platform Integrator), jonka tehtävänä on automatisoida TTA-prosessorin integrointi kohdealustalle, ja siten nopeuttaa suunnitteluprosessia. Diplomityössä toteutettiin alustaintegraattoriohjelmistokehyksen avulla kolme erillistä alustaintegraattorikomponenttia ja varmennettiin niiden toimivuus testisovellusta käyttäen.

Diplomityön toinen osuus dokumentoi TTA-prosessorien varmennusvuon. Työssä toteutettiin uusi varmennustyökalu TTA-prosessorin resurssien yksikkötestaamiseen, joka täydentää varmennusvuota. Varmennusvuon eri vaiheita hyödynnettiin alustaintegraattorien varmentamisessa.

PREFACE

The work for this thesis was done in the Department of Computer Systems at Tampere University of Technology as a part of the Function based platform (Funbase) project.

I would like to thank Prof. Jarmo Takala for giving me the chance to work on this interesting project and for his guidance and improvement ideas for this thesis. I am also the most grateful to Pekka Jääskeläinen, M.Sc., for his guidance and invaluable advices regarding this thesis and the work on TCE. In addition, I would like to express my gratitude to my present and former coworkers for creating such a pleasant working atmosphere and giving me a helping hand whenever it was needed. Especially, I would like to thank Teemu Pitkänen for sharing his hardware expertise.

I would also like to thank my family and friends for their support throughout my studies and life. Most of all, I would like to thank my dear Elina for her love and support.

Tampere, May 10, 2011

Otto Esko

CONTENTS

ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
CMOS	Complementary Metal Oxide Semiconductor
DMA	Direct Memory Access
DRE	Dead Result Elimination
DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EEPROM	Electronically Erasable Programmable Read Only Memory
FU	Function Unit
FPGA	Field Programmable Gate Array
GPP	General-Purpose Processor
HDL	Hardware Description Language
HIBI	Heterogeneous IP Block Interconnection
HW	Hardware
ILP	Instruction Level Parallelism
IP	Intellectual Property
ISS	Instruction-Set Simulator
JTAG	Joint Test Action Group
LE	Logic Element
LUT	Lookup Table
LSU	Load-Store Unit
Mbps	Megabits per second
MIMO	Multiple-Input Multiple-Output
N2H2	Nios II to HIBI version 2
PLA	Programmable Logic Array
PLD	Programmable Logic Device
RF	Register File
RISC	Reduced Instruction-Set Computer
ROM	Read Only Memory

RTC	Real Time Clock
RTL	Register Transfer Level
SRAM	Static Random Access Memory
SW	Software
SoC	System-on-Chip
SoPC	System-on-Programmable-Chip
TCE	TTA-based Co-design Environment
Tcl	Tool Command Language
TTA	Transport Triggered Architecture
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
VHDL	Very high speed IC Hardware Description Language
VLNV	Vendor, Library, Name, Version
XML	eXtensible Markup Language

1. INTRODUCTION

Reprogrammable logic devices, such as *Field Programmable Gate Arrays (FPGA)*, provide a fast and cost efficient way for testing and implementing custom digital designs. A variety of reusable *Intellectual Property (IP)* components allows the designer to create complex designs in reasonable time and synthesizable soft-core processors provide the possibility to implement functionality using software. It is often easier and faster to implement complex functionality using software rather than implementing the same functionality fully on hardware logic. However, while software adds flexibility and allows reconfigurability, *General-Purpose Processors (GPP)* are inferior in terms of performance and energy efficiency in comparison to fixed function hardware implementations.

Application-Specific Instruction-set Processors (ASIP) enable matching the flexibility of software and the performance of a fixed function hardware implementation by allowing the processor to be tailored to suit a specific program. In order to decrease the design time compared to a custom fixed function hardware implementation, an easily customizable processor architecture is needed, together with efficient design tools. *TTA-based Co-design Environment (TCE)* [1] fulfills these requirements by providing a toolset for customizing processors based on the modular *Transport Triggered Architecture (TTA)* template. However, previously, the TCE toolset only created an ASIP core and required the designer to manually integrate the core into the desired target platform. Manual integration was time consuming and error prone which hindered the design productivity.

For this thesis, a Platform Integration framework for TCE was created. The purpose of the framework is to automate the ASIP integration process to various platforms in order to decrease design time and to reduce the change of human error, thus increasing the design productivity. This thesis describes the requirements, design and implementation of the Platform Integration framework.

The second part of this thesis documents the verification flow of TTA-based ASIPs designed with TCE. In this thesis, a TTA Unit Tester tool is implemented to complete the verification flow with automated testing support for the processor components. The design and implementation of this tool are described in this thesis.

The thesis is divided into the following chapters. Chapter 2 introduces the basics of reprogrammable logic devices concentrating on FPGA devices. The concept of soft-core processors is also presented. Finally, System-on-Chip design and two different System-on-Chip design tools are described. Chapter 3 introduces the Transport

Triggered Architecture template which is used for ASIP design. The design environment and design flow of TTA-based ASIPs are also described in this chapter. The requirements, design and implementation of the Platform Integration framework are presented in Chapter 4. Chapter 5 documents the verification flow of TTA-based ASIPs and introduces the TTA Unit Tester tool implemented in this thesis. Chapter 6 verifies the Platform Integration framework functionality utilizing the verification flow. Finally, Chapter 7 concludes this thesis.

2. REPROGRAMMABLE LOGIC

Programmable Logic Devices (PLD) allow the end-user to configure the device to implement custom hardware designs unlike hard logic devices, such as *Application-Specific Integrated Circuits (ASIC)*, where the logic is fixed by the device manufacturer. First PLDs where implemented using *Read Only Memories (ROM)*. Later, *Programmable Logic Array (PLA)* devices were introduced. The PLAs consist of two levels of programmable logic: an AND-plane followed by an OR-plane. A common PLA configuration method was the fuse technology, where the unneeded connections are burned open like a fuse using an electric current. Due to the configuration technologies, both of these PLD implementations allowed only one time programming. [2] [3]

Reprogrammability was introduced by using *Electronically Erasable Programmable Read Only Memory (EEPROM)* or *Static Random Access Memory (SRAM)* to control the transistors on the chip. Furthermore, implementing SRAM is easier than implementing fuses on CMOS technology, which allows higher density devices. Over the years SRAM-based FPGAs have gained a strong foothold in the market. [2] [3]

2.1 Field Programmable Gate Array

Field programmable gate arrays are programmable logic devices. The term *field programmable* refers to the ability to "program it in the field", in other words, the programming can be done by the end-user [2]. The commonly used FPGA devices are reprogrammable but one time programmable FPGAs exist as well. One time programmability on CMOS is implemented using a so called antifuse technology [3].

In contrast to hard logic devices, the basic logic building blocks of FPGAs are not transistors or logic gates. Instead, an FPGA consists of programmable *Logic Elements (LE)* and programmable interconnections which allow the LEs to communicate with each other [3]. The anatomy of an LE can vary depending on the FPGA vendor and the device family, but quite often a reprogrammable LE is at least composed of a n-input *Lookup Table (LUT)* and a register [3]. For example, the logic element of the Altera Stratix II FPGA [4], which is called an *Adaptive Logic Module (ALM)*, has eight input combinational logic section (LUT), two adders, four multiplexers and two registers as illustrated in Fig. 2.1. These ALMs are stacked together to create bigger entities, *Logic Array Blocks (LAB)* and they form a two-dimensional array inside the device. The programmable interconnections can be

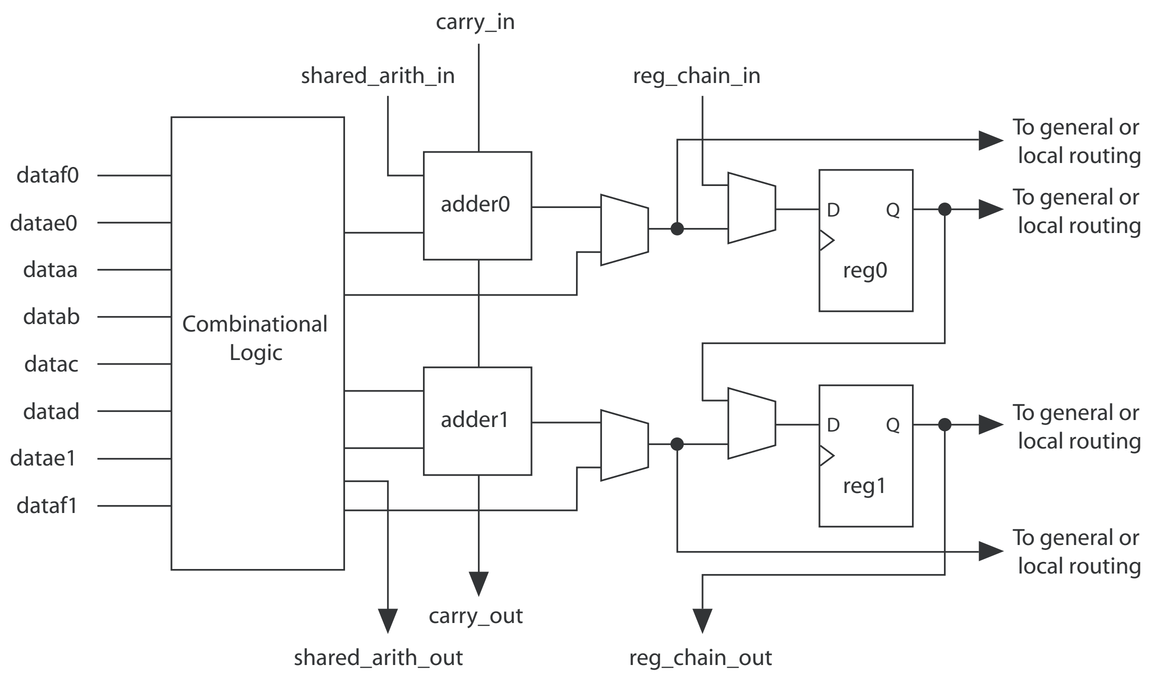


Figure 2.1: Adaptive Logic Module (ALM) is the basic building block of logic on the Altera Stratix II device family. The combinational logic block on the left consists of lookup tables and multiplexers. The horizontal signals are connected to interconnections and the vertical signals are connected to other ALMs. [4]

used to route signals from one LAB to another or to IO-blocks which are interfaced with the physical device pins. This kind of architecture is illustrated in Fig. 2.2.

In addition to programmable components (soft logic), an FPGA can include hard logic structures, such as general-purpose memory blocks or hardware multiplier elements. Implementing these kinds of elements on hard logic normally requires less area on the chip, consume less power and reach higher clock frequencies than their equivalent implementations on programmable logic. On the other hand, if the user design does not utilize these hard logic blocks, they just waste the silicon area on the chip. Over the years, there has been a variety of different hard logic blocks embedded in FPGAs, ranging from fully fledged hard processor cores to computational blocks which can be combined to support different data widths for *Digital Signal Processing (DSP)* applications. [3]

The key advantage of the FPGA is the flexibility provided by reprogrammability. Designs can be tested on the target hardware from the very early stages of the design process since the FPGA chips are standard off-the-shelf components which can be bought readily from the vendors. This improves the time-to-market of the product. In ASIC design, it might take months before the target hardware is available for testing because the chips need to be manufactured and tested during the design process of a product which increases time-to-market. Therefore FPGAs are often used as a test platform for ASIC designs. [3] [5]

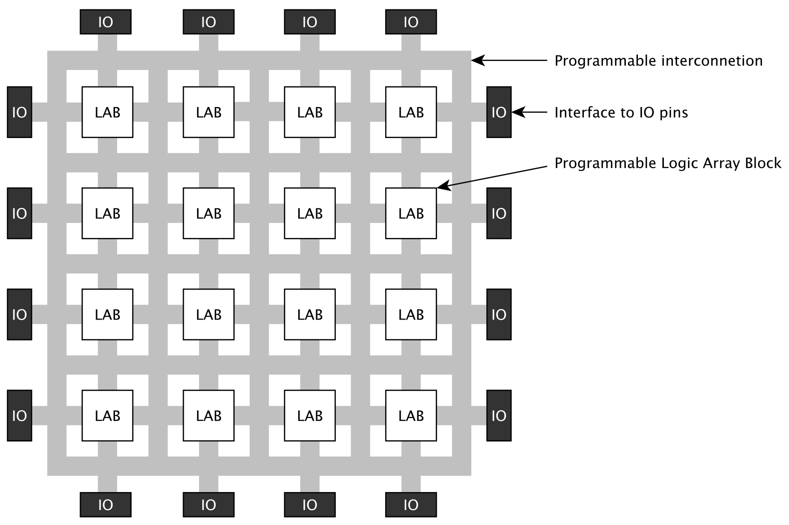


Figure 2.2: An example of an FPGA architecture. Programmable logic array blocks (LAB) are connected together via programmable interconnection network. On the outside edges, there are IO-blocks which are used to access device pins.

The reprogrammability of the FPGA can also be beneficial during the life cycle of a product. Hardware bugs can be fixed or new features can be added by upgrading the firmware of FPGA, which is easy and cheap in comparison to starting a new ASIC chip manufacturing iteration. However, the flexibility comes with a price. The reason FPGAs do not reign over ASICs, is that ASICs are superior in terms of the area, delay, power consumption and unit price in high volume products. Kuon and Rose have measured in their research [6] that on average, the FPGA implementations were 35 times larger, had 3.4 to 4.6 times lower clock frequencies and consumed 7.1 to 14 times more dynamic power than their equivalent ASIC implementations. On the other hand, the current trend shows that the ASIC manufacturing is coming more and more expensive which favors the use of FPGAs on low volume products. For smaller companies, FPGAs might be the only economically viable option for implementing custom digital designs. [3] [5]

2.2 Soft-core Processors

Soft-core processors are processors which are implemented on programmable logic such as an FPGA. Commonly, the soft-cores are written in *Hardware Description Languages (HDL)* which allows them to be synthesized on different FPGA technologies or even to be used in ASICs. The major FPGA vendors, such as Xilinx and Altera, provide soft-cores optimized for their own FPGA technologies. These soft-cores are also vendor specific: the typical end-user license restrict their use only for the products of the vendor. There are also open source soft-cores available such as the OpenRisc 1200 and the LEON3. [7]

In addition to providing the option to use soft-core processors, some FPGAs include hard processor cores. While the hard processors tend to be faster and smaller areawise, they have a few drawbacks. First of all, the number of the processor cores cannot be scaled to match the application. There might be too few or too many processors for the desired application. Also, the scalability, in terms of single thread performance, may not be achievable because the core architecture is fixed. Finally, the placement of the hard core on the FPGA chip is fixed which can lead to difficulties in interconnection routing between the core and other logic. [8]

Using soft-core processors allows the designer to include the exact number of processors needed by the application to the FPGA [8]. Furthermore, a soft-core gives the *CAD (Computer Aided Design)* tool more freedom to place and route the processor [8]. Soft-core performance can also be varied in some cases. For example, the Altera Nios II [9] soft-core is available in three different versions: economy, standard and fast. The economy version is a simple single cycle RISC (Reduced Instruction-Set Computer) core targeted for small size and offers very little customizability. On the other end, the fast version of the Nios II core is designed for high execution performance. For example, it has a six-stage pipeline and it includes a dynamic branch predictor. Naturally, the fast core implementation requires more logic elements from an FPGA than the economy core. In addition to the core variations, some of the core parameters can also be altered, such as the sizes of instruction and data caches and whether the core includes a hardware multiplier and/or a hardware divider. The Nios II even allows instruction-set extension with user specified custom instructions. [7] Customizability allows the designer to tailor the processor to meet the application demands better. Studies [8] [10] [11] show that varying the available design parameters allows performance scaling and FPGA resource scaling.

2.3 System-on-Chip Design

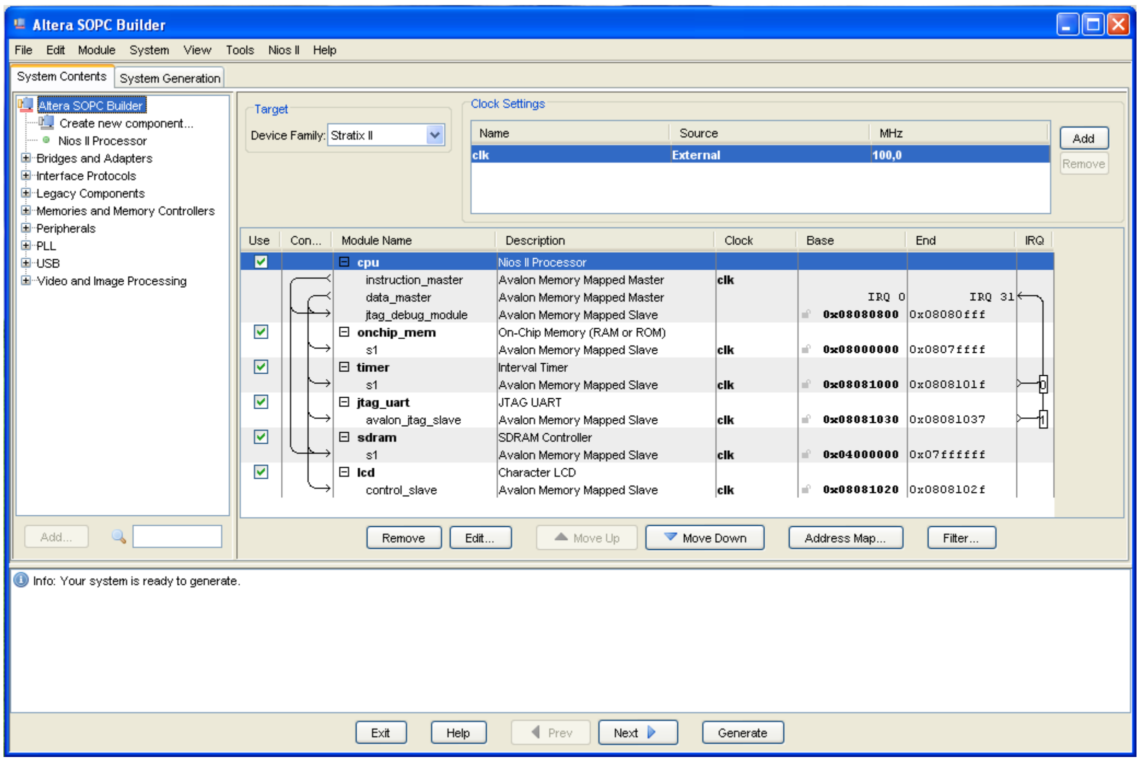
The increasing complexity of digital designs and the demand to increase the productivity has forced the design work from the *Register Transfer Level (RTL)* to a higher abstraction level. Over the years, System-on-Chip (SoC) designs have become a popular way to tackle the issue. The essential difference in the traditional RTL design and the SoC design is the size of basic building blocks: instead of logic gates and registers the designer can use complete component blocks. The range of such blocks can vary from computational components, such as a *Fast Fourier Transformation (FFT)* accelerator, to full fledged processor cores. These components are often referred as Intellectual Property (IP) blocks or cores. [12] [13]

One of the key factors in increasing the productivity is the hardware design reuse. The idea of hardware design reuse is by no means new, it has been done for decades. What makes the difference is the size of the reusable design blocks. The designer can integrate the pre-designed and pre-verified IP blocks together to construct complex systems in a modular fashion. The design time saved on reusing components rather than implementing them from scratch can significantly reduce the time-to-market of a product. Thus, selling and licensing reusable IP blocks has become a major business over the years. [12] [14]

Integrating the IP blocks together is an essential phase in the SoC design. In the past, the integration required more or less the use of "glue logic" implementations, which were custom-made hardware, to link the different components together to allow them to communicate with each other. The utilization of such communication glue logic has several disadvantages. First of all, the reusability of the components is degraded if they require a lot of integration work per use. The custom made communication hardware also needs to be verified on every new design. Moreover, the lack of standard communication interfaces makes the automated integration difficult. In order to overcome these problems, several standardized interconnection buses have been proposed, for example the AMBA (Advanced Microcontroller Bus Architecture) [15], created by ARM Ltd. The standard bus interfaces guarantee the integration compatibility between the different IP vendors. [12] [16]

One popular SoC design methodology is hardware/software (HW/SW) co-design where the functionality is partitioned between software and hardware component implementations. Naturally, a software implementation requires one or more processors to be included in the system. Compared to hardware, software implementations tend to be less efficient in terms of execution performance and power consumption. This is due to the overhead caused by the inherent need of the processor to fetch and decode instructions as well as load and store operands [5]. Software implementations are used because they add flexibility and increase the productivity since it is often easier to implement complex functionality using software rather than hardware. Also making changes later to the functionality tends to be easier in case software is used. For example, multimedia applications, such as video codecs, are often difficult to implement without some use of software as they might have to support lots of parameterizable options. State-of-the-art codecs might also require Digital Rights Management (DRM) software which is only available in executable binary form, thus compelling to use a processor. The key goal of HW/SW co-design is to partition the system in such a way that the system components are implemented in the most suitable way. The designer can, for example, use hardware accelerators to implement the computational intensive parts of a video codec and software to handle the complex functions and to control the encoding flow. [5] [16] [17]

The inherent reprogrammability of an FPGA adds flexibility to the SoC design. FPGAs can be used for rapid prototyping and verification of a SoC before the design is manufactured as an ASIC. Furthermore, FPGAs can be used in final products as well. In this case, the designs are sometimes referred as System-on-Programmable-Chip (SoPC) designs. A variety of available soft-core processors makes the FPGA



Device Family: Stratix II

Name	Source	MHz
clk	External	100,0

Use	Con...	Module Name	Description	Clock	Base	End	IRQ
✓		cpu	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk			
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave		0x08080800	0x08080fff	
✓		onchip_mem	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk	0x08000000	0x0807ffff	
✓		timer	Interval Timer				
		s1	Avalon Memory Mapped Slave	clk	0x08081000	0x0808101f	0
✓		jtag_uart	JTAG UART				
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk	0x08081030	0x08081037	1
✓		sdram	SDRAM Controller				
		s1	Avalon Memory Mapped Slave	clk	0x04000000	0x07ffffff	
✓		lcd	Character LCD				
		control_slave	Avalon Memory Mapped Slave	clk	0x08081020	0x0808102f	

Info: Your system is ready to generate.

Figure 2.3: The main view of Altera SOPC Builder tool. The dialog in the middle shows the IP components and interconnections of the designed system.

an interesting platform for SoC development not only for commercial but also for educational purposes.

2.3.1 Altera SOPC Builder

SOPC Builder [18] is a system-level design tool for System-on-Programmable-Chip designs from Altera. The tool is mainly targeted for developing Nios II soft-core-based systems, but it can be used for designs with or without processors. The SOPC Builder provides a graphical user interface for the designer to create a system by adding and connecting IP components from the IP library of the tool. The main view of the SOPC Builder tool is presented in Fig. 2.3. The tool automatically integrates the components together by generating an Avalon [19] interconnection network between the components and creates a top level HDL implementation of the design. The SOPC Builder does not have the capability to synthesize the design, instead it relies on Altera Quartus II [20] to perform this task. The use of SOPC Builder is tied to Altera devices.

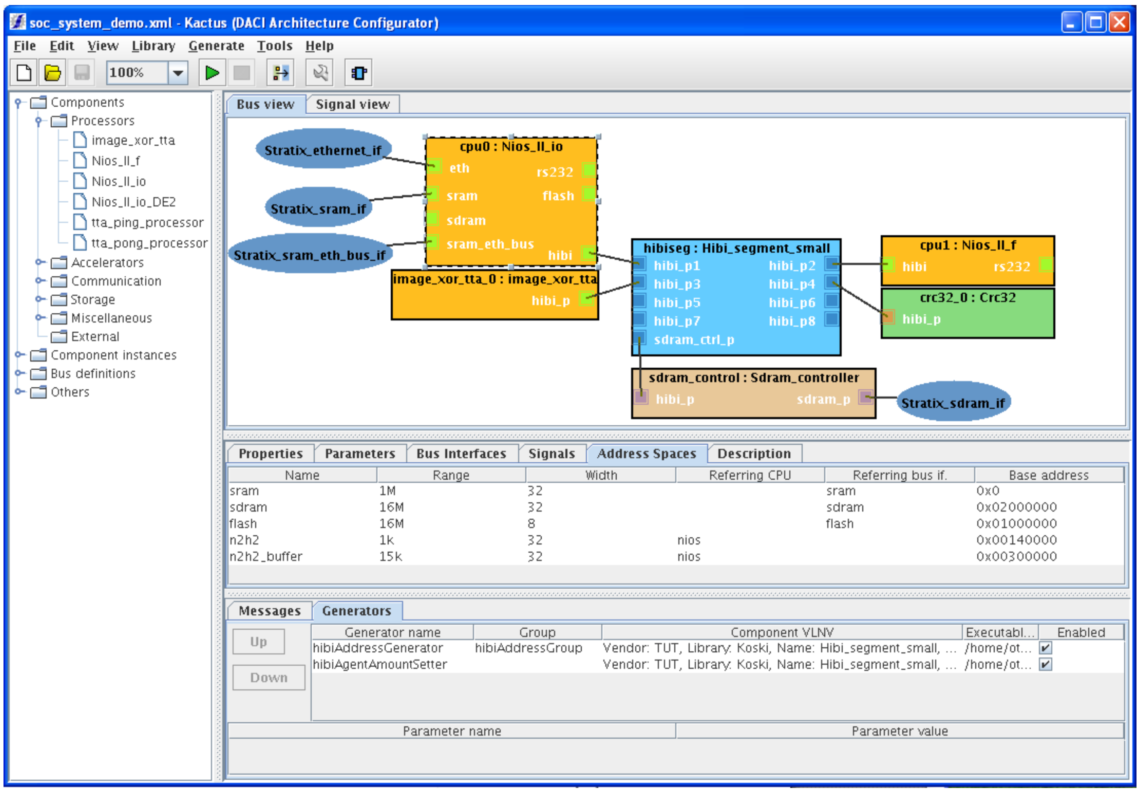
The main bus interfaces in SOPC Builder are the Altera Avalon Interfaces [19] which consist of Avalon Memory Mapped (Avalon-MM), Avalon Streaming (Avalon-ST), Avalon Memory Mapped Tristate, Avalon Clock, Avalon Interrupt and Avalon Conduit interfaces. The Avalon-MM provides a typical memory address-based read-write interface for master-slave-connections. It is perhaps the most commonly used interface in the SOPC Builder systems. The Avalon-ST is a unidirectional source-sink-type interface for low latency, high throughput streaming purposes. The Avalon Memory Mapped Tristate interface resembles the Avalon-MM but it is designed for off-chip tristate connections. The Avalon Conduit interface is used for exporting arbitrary signals outside of the SOPC Builder system. The exported signals can be connected to other on-chip designs or to off-chip components via the FPGA pins. The Avalon interrupt interface describes the interrupt senders and receivers. The Avalon clock interface is used to associate a clock and a reset signal to components and their interfaces. The Avalon Interface specification does not restrict the number of different interfaces a single component can have. It is also possible for a component to include multiple instances of the same type interface.

The automatic integration of arbitrary components requires some metadata of these components. SOPC Builder uses Hardware Description Files (_hw.tcl) written in Tcl (Tool Command Language) for this purpose. This file declares the general component information, such as the name, version, static or user modifiable parameters and, most importantly, the Avalon Interfaces included in the component. The interface declaration maps the actual design signals to the logical bus signals and describes their width and direction. This signal mapping metadata allows the tool to automatically integrate the components together without enforcing strict naming conventions of the bus interface signals in the components. [18]

2.3.2 Koski Framework

Koski [21] is a system design framework created at Tampere University of Technology (TUT). Koski is targeted for multiprocessor SoC system-level modeling and rapid FPGA prototyping. It also supports design space exploration. One of the distinct features of Koski is that the system can be described and modeled using the *Unified Modeling Language (UML)* with an extension profile designed for embedded hardware modeling. It is also possible to model the system in a more traditional way by using a graphical tool called Kactus which is illustrated in Fig. 2.4. The Koski design flow is controlled using the Koski GUI tool which allows the designer to select and configure the phases of the flow. [21] [22]

Koski utilizes IP-XACT [23] to describe the metadata of the system components. The IP-XACT standard was originally developed by the SPIRIT Consortium to provide a language independent metadata specification for *Electronic Design Automation (EDA)*. The metadata itself is stored using the *eXtensible Markup Language (XML)*. IP-XACT can be used for example to define components, such as IP blocks or processors, and bus interfaces, which define the interfaces between components. A bus interface definition describes the names, directions and widths of the logical signals associated to the interface. A component definition containing a bus inter-



Name	Range	Width	Referring CPU	Referring bus if.	Base address
sram	1M	32		sram	0x0
sdram	16M	32		sdram	0x02000000
flash	16M	8		flash	0x01000000
n2h2	1k	32	nios		0x00140000
n2h2_buffer	15k	32	nios		0x00300000

Generator name	Group	Component VLNV	Executabl...	Enabled
hibiAddressGenerator	hibiAddressGroup	Vendor: TUT, Library: Koski, Name: Hibi_segment_small, ...	/home/ot...	✓
hibiAgentAmountSetter		Vendor: TUT, Library: Koski, Name: Hibi_segment_small, ...	/home/ot...	✓

Parameter name	Parameter value

Figure 2.4: The main view of the Kactus tool which is part of the Koski framework. Kactus is used to model the hardware system, as shown in the top right main dialog.

face then describes how the actual component signals are mapped to the logical bus signals of the interface. In addition, the component definition describes the design files needed to synthesize the component. Every IP-XACT definition also has a *VLNV (Vendor, Library, Name, Version)* description which is used to identify the components. Koski uses this metadata to integrate the components and to create a HDL description of the system. [22] [24]

The main bus interface used in Koski is the *Heterogeneous IP Block Interconnection version 2 (HIBI)* [25] which was designed and implemented at TUT. HIBI is intended for integrating IP blocks together and it is designed to be a topology-independent, scalable and yet high-performance connection network. HIBI can be constructed in a modular fashion by using HIBI wrappers which are parameterizable HW components. The HIBI wrappers implement a distributed arbitration of the shared bus, and the wrappers can be used to implement transition from one clock domain to another. The IP blocks can be directly attached to these wrappers or there can be smart adaption blocks in between. An example of a such smart adaption block is the Nios II to HIBI version 2 (N2H2) adapter [26] which was created to attach the Nios II processor to HIBI and to implement direct memory access (DMA) transfers via HIBI. The N2H2 adapter implements an Avalon-MM slave interface, which the Nios II uses to configure the adapter to send and receive DMA transfers, and an Avalon-MM master interface, which allows the adapter component to gain direct access to the memory. On the other side, the adapter is interfaced with a HIBI wrapper. [25] [26]

Unlike the SOPC Builder, Koski is platform independent and can target different platforms and synthesis tools. The modular design of Koski allows 3rd party tools to be integrated with Koski. When, for example, the designer uses a Nios II processor in his design, the software compilation for the Nios II can be executed from the Koski GUI. Similarly, the Koski GUI can be used to create Quartus II configuration settings for easy synthesis of the design. [21] [22]

3. APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSORS

Application-specific instruction-set processors are processors which are tailored to execute a single application or a set of applications from an application domain. The general-purpose performance of the processor can be put aside and the tailoring effort is focused on the specific purpose. Three key factors in processor tailoring are the performance, area and power consumption. Compared to a general-purpose processor, an ASIP can achieve higher performance if the processor resources, such as registers and computational units, are matched to the application at hand. The area usage can be optimized as well. For example, if the application does not utilize the division operation, the divider unit can be removed from the processor architecture. Power consumption is related to the performance and area. Static power consumption is directly proportional to area, in other words, the number of transistors. [5] [10] [27]

One of the key motivations for using ASIPs is to increase the application performance without the need to implement or purchase complex fixed function hardware components. Manual IP block design can often be time consuming, and, thus, expensive. An ASIP implementation fits between a GPP software implementation and a pure HW implementation in terms of performance and design time. ASIP implementations can be scalable in terms of *performance per area* and *performance per power consumption* factors. [5] [10]

In order to save design time compared to a fixed function HW implementation, there must be an easy and fast method for the design of ASIPs and their software toolchains. Starting to create a new processor architecture and a compiler from scratch is out of the question as it would be too time consuming. A customizable processor architecture is therefore needed. The following sections will describe one such architecture and a design toolset for implementing ASIPs using the architecture.

3.1 Transport Triggered Architecture

Instruction Level Parallelism (ILP) describes the potential to execute instructions simultaneously [28]. In other words, if the instructions are not dependent on each other, they could be evaluated concurrently. Exploiting the ILP has been one of the key techniques in processor performance improvements over the years. There are two main categories in exploiting the ILP. The first approach relies on hardware to dynamically find the parallel instructions during run time. For example, the modern desktop PC processors use this approach. Naturally, this requires the inclusion of a complex hardware logic for searching the parallel instructions which increases processor size and power consumption. The other option is to depend on the software compiler or the programmer to statically find the parallel instructions before run time. In this approach, the performance depends heavily on the compiler. [5] [28]

Very Long Instruction Word (VLIW) processors are known for their ability to exploit static instruction level parallelism thanks to their parallel *Function Units (FU)* [28]. A VLIW instruction consists of multiple operations which are executed concurrently in different function units. These operations are scheduled statically during compile time which means a VLIW processor does not need to include a complex instruction dependency detection hardware logic which simplifies the processor implementation.

However, the scalability of a traditional VLIW is limited due to bottlenecks in the architecture. The first bottleneck forms into the *Register File (RF)*. In the worst case scenario, every function unit must read and write to the register file on the same clock cycle. If the architecture has N function units which each have two input ports and one output port, the register file must have $2N$ read ports and N write ports. The complexity, and, thus, the size of the register file increases at least in a linear fashion as a function of the RF port count. Another bottleneck comes from the bypassing network between the function units which allows the results from one FU to be directly written to the input of another FU without circulating the result through the register file. If the bypassing network is fully connected, which means that the FU output ports are connected to all of the input ports, the network complexity grows in a quadratic fashion as the number of FUs increase. [29]

Transport Triggered Architecture (TTA) template is based on the VLIW and provides a solution to the bottlenecks of the traditional VLIW. TTA changes the traditional operation based programming paradigm into an operand transport based programming paradigm. This means that a TTA instruction does not describe the executed operations but the performed operand transports. The operations themselves are executed as a side effect of these operand transports. Thus, the TTA instruction controls the connection network between the function units and the register files rather than the function units and register files themselves. [29]

3.1.1 Hardware Characteristics

As the TTA instruction controls the *InterConnection network (IC)*, it means that the IC connections are visible to the programmer. Thus, the TTA can be described as an *exposed datapath architecture*. The programmer-visible IC network allows the processor designer to customize the IC connections because the programmer is aware of the connections and can only use the existing connections. Thus, the IC customiz-

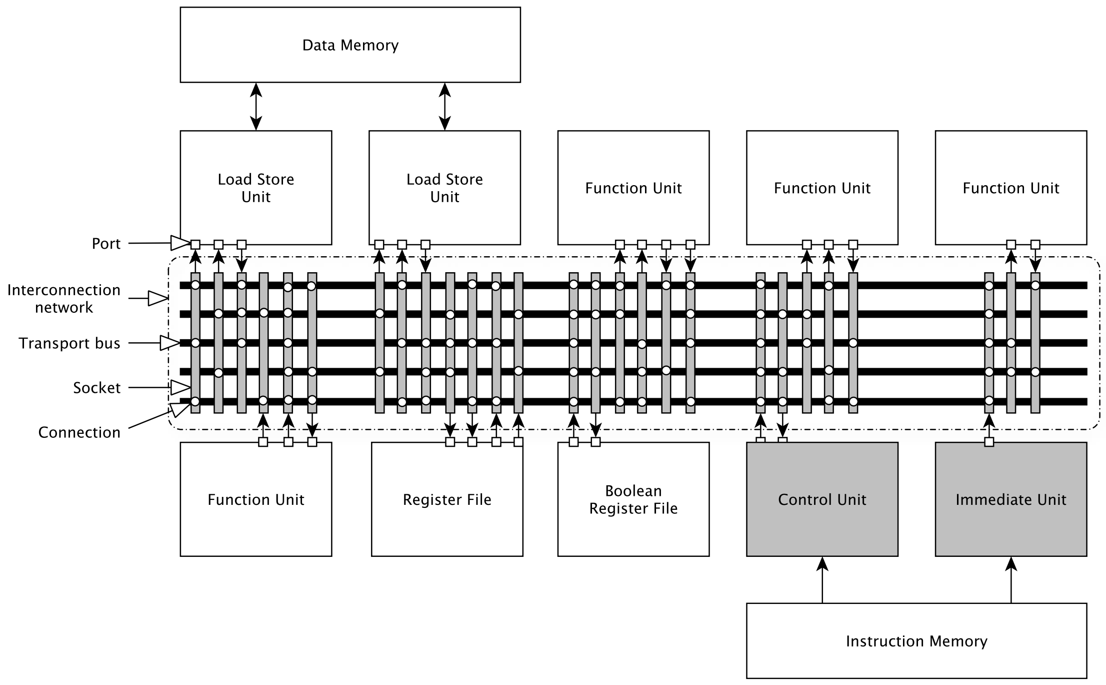


Figure 3.1: Example of a Transport Triggered Architecture (TTA). Interconnection network, which connects Function Units and Register Files together, is illustrated in the middle.

ability allows the processor designer to control the IC network complexity. [29]

An example of a TTA processor is presented in Fig. 3.1 which shows the IC network in the middle. The IC network consists of transport buses, sockets and connections between them. The transport buses are used for transferring the operands. Each bus can issue one transfer per instruction and the number of buses can be customized. Sockets are used to connect the function unit ports to the transport buses. [29]

TTA function units can implement one or more operations and they can be internally pipelined. An operation is executed by issuing the operand transports to the input ports of an FU. One of these ports is a so called *trigger port* which has a special purpose. When an operand is transferred to this port, the operation execution is triggered. After the time defined by the static latency of the operation, the result can be read from the output port of the FU. The FU ports are typically registers which means that the values are stored in the ports until they are overwritten by the next operation. This helps to reduce the register file traffic [30].

The register files in TTA do not differ much from the function units. The RFs are connected to the IC in a similar way, and,thus, the available RF ports and their connections are visible to the programmer. Thanks to the exposed datapath the processor, the designer has the freedom to decide how many ports there are in each register file and the application code must adapt to the available resources. This helps in controlling the RF complexity. Besides the RF port count, the processor designer can choose the number and the size of the registers there are in a single RF, and, of course, the number of register files there are in the processor architecture. For example, a huge multiported RF can be divided into several small and simple register files. [31]

In addition to customizing the register files, the TTA template allows the function units to be modified as well. The number of the function units can be changed and the operations implemented in each unit customized as well. The number of transport buses can also be modified. As the Fig. 3.1 illustrates, a TTA processor is constructed in a modular way which makes adding and removing components straightforward. [29]

3.1.2 Programming Model

TTA has only one actual instruction: *move*, which implements operand transports. The need to support only a single instruction combined with the static operation scheduling makes the control logic of TTA very simple.

The operand transports of TTA are also visible to the programmer. In a traditional operation based assembly, an addition $r3 = r1 + r2$ could be executed as:

```
add r3, r1, r2
```

In TTA assembly, the addition is executed by defining the operand transports:

```
r1 -> add.o1
r2 -> add.t
add.r -> r3
```

Or, if there are multiple transport buses available, the input operands could be transferred simultaneously:

```
r1 -> add.o1, r2 -> add.t
add.r -> r3
```

It is notable that the operation latency is visible to the programmer. In the example above, the latency of the *add* operation is 1 clock edge. The operation result can be read from the output on the next instruction after the triggering move.

One of the advantages of the operand transport paradigm is software bypassing [30] and *Dead Result Elimination (DRE)*. For example, the following code in operation based assembly:

```
add r3, r1, r2
shift r6, r3, r4
store r3, r6
```

could be executed on a TTA with two transport buses as:

```
r1 -> add.o1, r2 -> add.t
add.r -> shift.o1, r4 -> shift.t
add.r -> store.o1, shift.r -> store.t
```

As the example demonstrates, the *addition* result can be bypassed to the *shift* and *store* operations. Likewise, the *shift* result is bypassed to *store*. Furthermore, as the results of *addition* and *shift* can be bypassed to the next instructions, dead result elimination removes the unnecessary result writes to the register file. In the example, these optimizations decreased register file utilization by removing two register writes and reads.

TTA also makes it easy to schedule code for custom *Multiple-Input Multiple-Output (MIMO)* operations. The greatest advantage in case of MIMO operations is that all the input and output operands do not have to be transported in a single instruction cycle. The input port registers of a function unit hold their values until they are overwritten and the operation results stay in the output port registers as long as the next operation is triggered and the new results are written to the outputs. This makes it possible to divide the operand transports to multiple instruction cycles which puts less pressure on the register file. For example, if there was an operation with eight inputs and all of the operands were stored in a register file, there would have to be eight read ports in a VLIW register file in order to provide all the operands in a single instruction cycle. However, in case of TTA, the input operands can be transported in multiple instruction cycles depending on the available RF read ports. If there are, for example, two read ports, the input operands can be transported in four instruction cycles.

3.2 TTA-based Co-design Environment

TTA-based Co-design Environment (TCE) [1] is a toolset for implementing application-specific processors based on the TTA processor template. The main use case of the toolset is rapid co-design of processor based accelerators with minimal manual RTL coding. TCE toolset is developed at the Tampere University of Technology.

The most essential tools in TCE are the processor designer tool *ProDe*, a retargetable high level language compiler *tcecc*, the retargetable *Instruction-Set Simulators (ISS) ttasim* (command line version) and *proxim* (graphical user interface version) and the processor generator *ProGe*. These tools allow high level language (HLL) to RTL design flow. The retargetability of the tools means that they automatically adapt to the processor architecture at run time. Most of the design work with TCE is done on the architecture level which makes the retargetability a key factor in reducing the design time, and, thus, increasing design the productivity. The design process is discussed in more detail in Section 3.3.

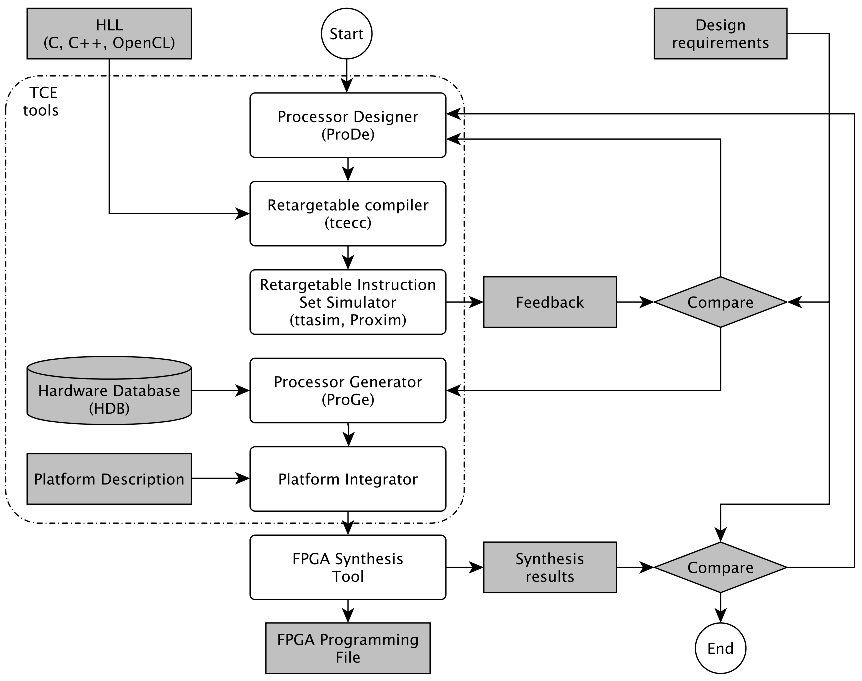


Figure 3.2: TCE design flow

TCE allows the designer to customize TTA processors. The designer can, for example, change the number of function units in the architecture, modify which operations are included in a function unit and even create new operations. The register files can be customized as well. The designer can change the register width, the register file size and the register file port count. The number of register files is also customizable. The interconnection network of the processor can be tailored as well. The number of transport buses and the connections between the FUs, RFs and transport buses are also modifiable.

3.3 ASIP Design with TCE

The goal of the TCE ASIP design flow, described in [32], is to produce a processor which is able to execute a specific application while complying with the restrictions set by the design requirements. TCE ASIP design flow is illustrated in Fig. 3.2. The design flow inputs are the HLL source code of the desired application and the design requirements. These requirements can, for example, define the amount of FPGA resources the implementation can use, the target execution time or performance, the minimum clock frequency or the maximum allowed energy consumption. At the beginning of the design flow, the designer uses *ProDe* to create a starting point architecture. *ProDe* stores the processor architecture description in XML-format to an *Architecture Definition File* (ADF). Alternatively, a pre-made ADF can be used as the starting point.

The next step is to compile the source code for the starting point architecture with the *tcecc* compiler which outputs a *TTA Program Exchange Format* (TPEF) binary file. The retargetable instruction-set simulator *ttasim* gets the ADF and TPEF files as input and produces simulation results, such as execution cycle count, processor resource utilization data and optionally execution traces which can be utilized to extract profiling data. These simulation results are the feedback from the flow. The designer analyses the feedback and modifies the architecture accordingly with *ProDe*. Then a new iteration is started and the new feedback shows how the modifications affected the results. This iteration process is also known as manual processor *Design Space Exploration (DSE)* [33]. TCE also includes an experimental *explorer* [33] tool which can be used to automate the processor design space exploration. In the automated DSE, the designer sets goals for the exploration and then the explorer modifies the processor architecture until the given goals are reached.

Processor design space exploration is continued until the design requirements are met or the designer determines that the results are adequate. It should be noticed that at this point the maximum clock frequency of the processor is unknown, and thus, the actual run time of the application is also unknown because the simulation results only tell the instruction cycle count. If the design requirements set a target clock frequency, one can determine the maximum instruction cycle count. In order to find out the actual maximum clock frequency, the execution time and the FPGA resource usage, the processor must be implemented and synthesized. The RTL implementation of the processor is generated with the *ProGe* tool. Before this can be done, the processor architecture resources, namely the FUs and RFs, must be mapped to their actual RTL implementations. The FU and RF implementation are stored in *Hardware Databases (HDB)*, and the resource mapping is done by simply defining which HDB entry is used for each architecture resource. The mapping information is written to an *Implementation Definition File (IDF)*. When the mapping is done, *ProGe* uses the ADF and IDF to create the processor RTL implementation by generating the IC network and connecting the FUs and RFs together.

At this point, only the processor core has been implemented. In order to execute applications, the core must be interfaced with the target platform. This can be done with *Platform Integrator* which was implemented for this thesis. The main idea of the *Platform Integrator* is to interface the core with the memory components and the FPGA board and to create synthesis tool project files to allow easy integration. *Platform Integrator* is presented in more detail in Section 4.

After the integration, the processor can be synthesized with a 3rd party synthesis tool. The synthesis produces an FPGA device programming file which is used to configure the design onto the FPGA for execution. The synthesis results are valuable feedback for the design flow. The results determine, for example, the actual resource usage and maximum clock frequency of the processor. The designer com-

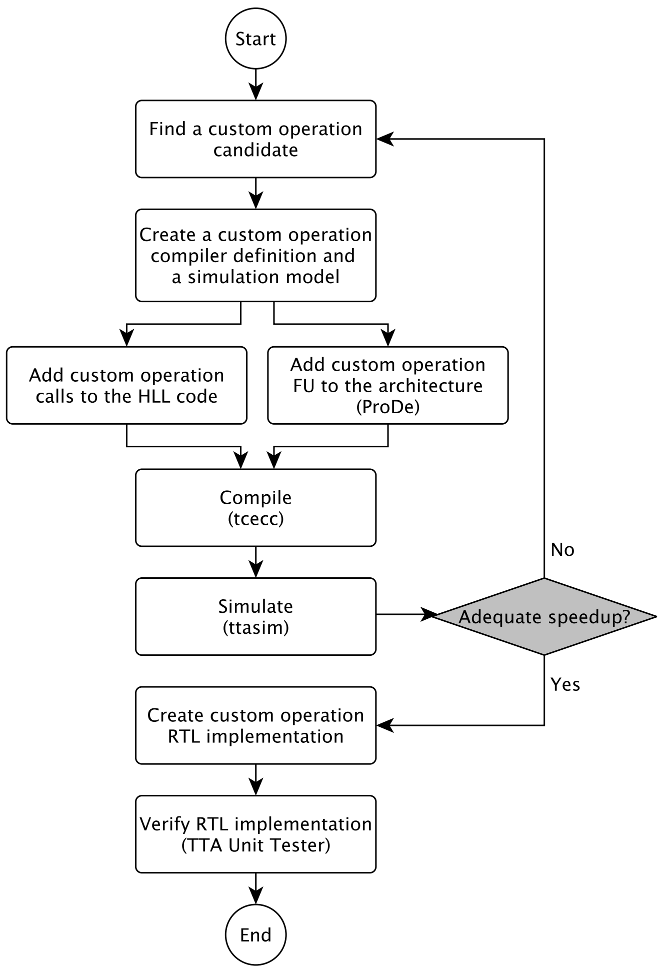

Start
Find a custom operation candidate
Create a custom operation compiler definition and a simulation model
Add custom operation calls to the HLL code
Add custom operation FU to the architecture (ProDe)
Compile (tcecc)
Simulate (ttasim)
Adequate speedup?
No
Yes
Create custom operation RTL implementation
Verify RTL implementation (TTA Unit Tester)
End

Figure 3.3: TCE custom operation design flow.

pares these results with the design requirements to determine whether the iterative design process can be finished.

In addition, TCE allows the designer to exploit custom hardware operations which can be used to accelerate the application. TCE allows the custom operations to be tested and evaluated without having the RTL implementation of the custom operation, thanks to the separation of the processor architecture and implementation.

The custom operation design flow, described in [32], is illustrated in Fig. 3.3. The flow begins by searching for a custom operation candidate. Application profiling can be helpful for this purpose. When a candidate is found, the designer creates a *custom operation compiler definition* by using the Operation Set Editor tool *OSEd*. The compiler definition simply describes the name of the operation and the number of input and output operands. In order to simulate the operation, a *simulation model* is needed. This model implements the operation behavior using C/C++. Usually the software function in the accelerated program can be exploited in defining the simulation model. If, for example, the custom operation replaces a function, the function code can be utilized as the simulation model.

Using the new custom operation requires modifications to the processor architecture and the HLL source code. First, the designer must add a function unit containing the custom operation to the architecture. At this point, the custom operation latency must be defined. Sometimes it can be difficult to determine the operation latency before the operation hardware is implemented. The designer can either take an educated guess or change the latency between iterations to find out which latencies would be feasible.

For the software to be able to use the custom operation, the designer needs to modify the source code to utilize the new operation. This is done by calling the operation via TCE-specific *operation macros* or *intrinsics*. In case the custom operation implements a function from the original code, these function calls are replaced with calls to the operation macros. When the modifications are ready, the application can be compiled and simulated. Feedback from the simulation will reveal how the custom operation affected the execution cycle count. If the results were not satisfying, another custom operation can be tested by starting a new iteration.

If the custom operation speedup was adequate and the designer chooses to include the custom operation to the architecture, the custom operation must be implemented. This is the only step in the TCE design flow where the designer is required to write RTL code, but then again, using custom operations is not compulsory. The designer can use utilize TTA Unit Tester, which was created for this thesis, to assist in the implementation of the custom operation. TTA Unit Tester verifies that the RTL implementation of the custom operation is equal to its simulation model. The TTA Unit Tester tool is described in more detail in Section 5.2. When the custom operation implementation is ready, the custom FU is added to a hardware database with the *hdbeditor* tool. This allows the FU to be reused in later designs.

4. PLATFORM INTEGRATION FRAMEWORK

The processor generator tool of TCE only creates the processor core. This processor core needs to be integrated into the target platform before the design can be executed. An effortless integration process is important for a fluent ASIP design flow because it allows the synthesis preparations to be made quickly and, thus, reduce the time to acquire synthesis results.

Previously, the TTA core integration required manual effort of the designer, but for this thesis a Platform Integration Framework was created to automate the integration flow.

4.1 Requirements

There are two main use cases for the *Platform Integrator (PI)*. The first one is to integrate the TTA processor straight to the targeted FPGA board, in other words, implement a stand-alone TTA on an FPGA. This option is discussed in more detail in Section 4.4. The second option is to wrap the TTA as an IP-block which can be used in System-on-Chip designs. This case is further discussed in Section 4.5. Both of these use cases share the same basic integration steps which are:

1. Create a wrapper around the TTA core.
2. Connect memory components to the TTA core. In case on-chip memory is used, the memory components need to be created and instantiated inside the wrapper.
3. Export all unconnected signals out of the wrapper to allow external connections. These signals can include, for example, control signals, such as the *clock* and *reset*, bus interface signals, peripheral signals and interface signals to off-chip devices such as memory devices and a *Digital-to-Analog Converter (DAC)*.
4. Perform platform specific tasks. For example, map the exported signals of the processor wrapper to FPGA pins or bus signals.
5. Write project files or metadata files for 3rd party tools. This step makes it easier to utilize TTA designs in other EDA tools.

The operation principle of the Platform Integrator is illustrated in Fig. 4.1.

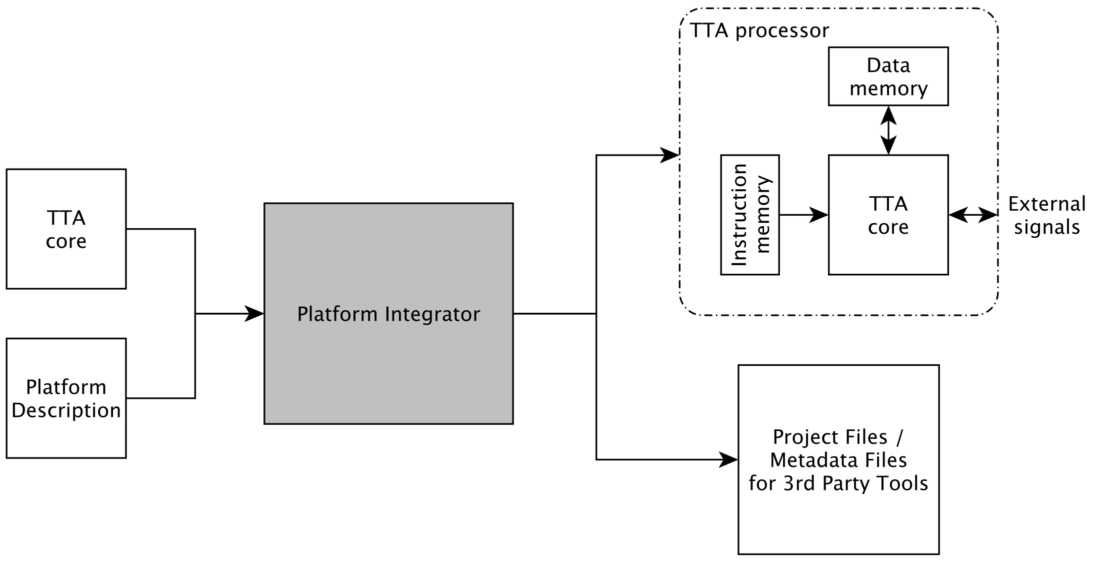


Figure 4.1: The operation principle of the Platform Integrator. The Platform Integrator connects the TTA core to memory components and conduits external signal interfaces out of the TTA processor wrapper (presented as a dashed rectangle). In addition, the tool-specific project or metadata files of the TTA processor are generated to allow the design to be used in 3rd party tools.

Platform Integrator needs to be able to implement the mentioned steps in an automated fashion. Furthermore, *PI* should be made vendor independent in such a way that multiple vendors can be supported. This means that the vendor specific handling should take place on the lowest possible level of the framework to make it easy to extend the vendor support later on.

The hardware support for different platforms is realized by creating platform specific Hardware Databases. These HDBs contain FU and RF implementations specifically targeted for the platform in question. A platform specific HDB can, for example, contain an LSU which includes an optimized memory controller for the SDRAM chip available on a particular FPGA board.

4.2 Implementation

Platform Integrator was not implemented as a separate tool but it was integrated into the *ProGe* [34] tool. *PI* is comprised of three abstract classes shown in Fig. 4.2.

PlatformIntegrator is the main class of *PI*. It contains the information on the specific platform. Integration is performed by calling method *integrateProcessor()*. This method takes the TTA core created by *ProGe* as a parameter. The TTA core is given as a *NetlistBlock* which is an object model containing the ports and parameters of the hardware implementation. *PlatformIntegrator* creates a new *Netlist* object for the TTA core and other integrator components. A *Netlist* models the connections

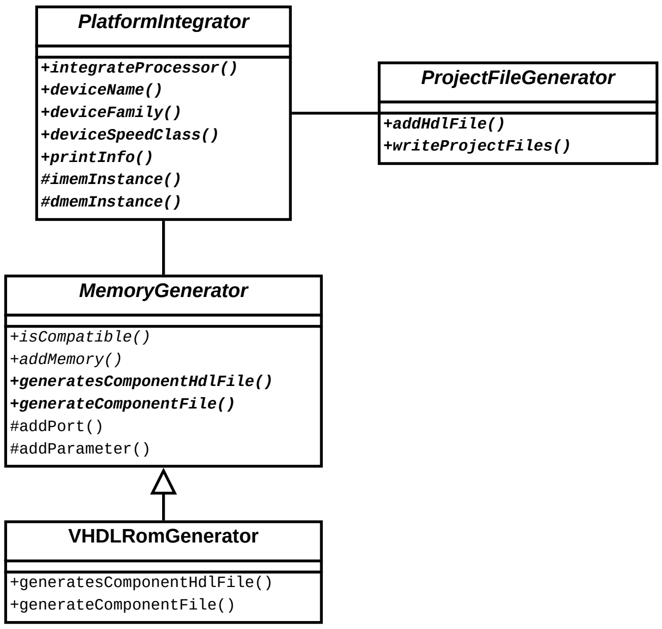


Figure 4.2: Main classes of the Platform Integrator Framework

between the ports of the different *NetlistBlocks*.

PlatformIntegrator class also includes methods for querying information about the FPGA device such as *deviceFamily()*, *deviceName()* and *deviceSpeedClass()*. Method *printInfo()* is used to output basic information about the Platform Integrator component for the designer. Protected methods *dmemInstance()* and *imemInstance()* are used to get appropriate *MemoryGenerator* object instances which are supported by the platform and compatible with the given TTA core.

MemoryGenerator class is responsible for connecting a memory to the TTA core. Depending on the memory type and setup, *MemoryGenerator* might need to create and instantiate a memory component or a memory controller and to create the needed connections. The different memory setups are discussed in more detail in Section 4.3. Method *isCompatible()* is used to determine whether the TTA core has a compatible memory interface for the specific memory generator. If the core is compatible, the method *addMemory()* creates the needed signals, instantiates the memory component or a memory controller and connects it to the TTA core in the *Netlist* given as a parameter. In addition, there are methods *generatesComponentHdlFile()* and *generateComponentFile()*. The first one is used to determines whether the *MemoryGenerator* creates HDL files and the latter is utilized to create these files.

One *MemoryGenerator*, *VHDLRomGenerator*, was realized for the base implementation of the Platform Integrator. This memory generator implements a simple read-only instruction memory where the memory contents are stored as a VHDL array of *std_logic_vectors*. This lets the synthesis tool to decide how to implement the actual memory and allows the tool to perform memory size optimizations.

VHDLRomGenerator can be used on every platform, hence it was added to the base implementation.

ProjectFileGenerator is the base class for creating project or metadata files from the design. Commonly, the design HDL files need to be listed in the project and metadata files and therefore, the class has method *addHdlFile()* for this purpose. *ProjectFileGenerator* has access to the *PlatformIntegrator* which allows it to query device specific information straight from the *PlatformIntegrator*. The project file generation is started by calling the method *writeProjectFiles()*.

4.3 Memory Interfacing Considerations

Interfacing the TTA core with a memory can be done in different ways. Figure 4.3 distinguishes the four different data memory connection types supported by the *MemoryGenerator*. The first two cases on the left use an on-chip memory. In these cases, the *MemoryGenerator* generates and instantiates the memory components inside the processor wrapper. The difference between these two options is the location of the memory controller component. On the very left, the memory controller is integrated in the load-store unit of the TTA core which means that the external interface of LSU contains low level memory control signals. The second option from left in Fig. 4.3 illustrates the situation where the memory controller is outside of the TTA core in the processor wrapper. The situation is quite similar to the first case with the exception that now the *MemoryGenerator* is also responsible for creating and instantiating the memory controller component. In this case, the LSU can contain a higher level memory interface. For example, it could only define the memory operation and the memory address and the memory controller converts them to low level memory control signals.

The two cases on the right in Fig. 4.3 differ from the first two only by using an off-chip memory instead of an on-chip memory. Again, the memory controller component can either be integrated in the LSU or be created by the *MemoryGenerator*. It should be noticed that the memory controller can, for example, control a cache or even a cache hierarchy. Caches are often desired when off-chip memories are used since the memory access latency is longer compared to an on-chip memory.

The instruction memory interfacing differs a bit from the data memory interfacing. The control unit of TTA is responsible for fetching the instructions from the memory and the fetch unit contains a rather simple memory controller. The fetch unit assumes that an instruction can be fetched with a single load operation. This can usually be implemented with an on-chip memory, but the long instruction word can be a problem with off-chip memories. If the data bus width of the off-chip memory is smaller than the instruction word width, the *MemoryGenerator* needs to generate an additional memory controller between the off-chip memory and the fetch unit. This memory controller reads multiple memory locations to provide a

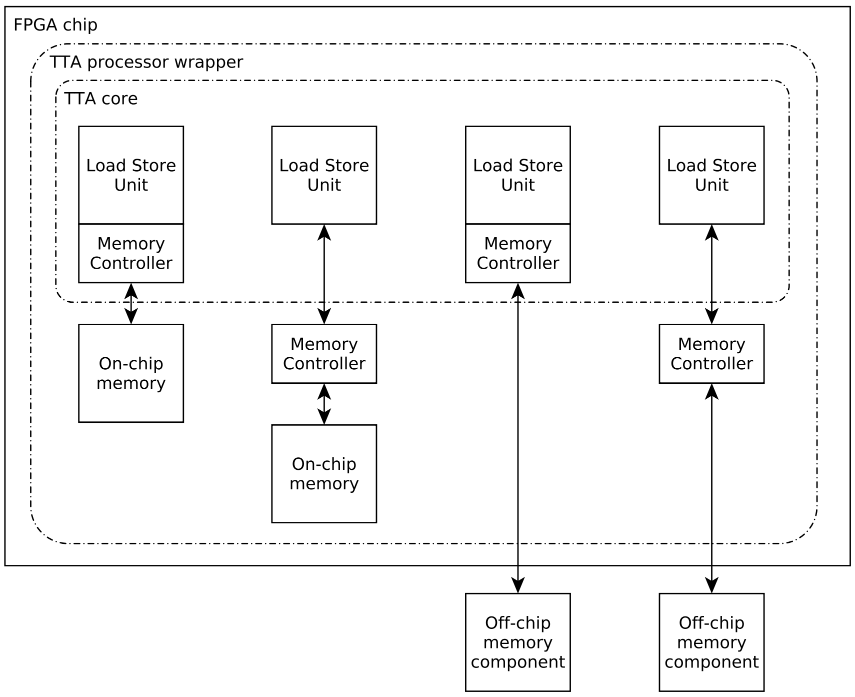


Figure 4.3: Different ways of connecting a data memory to the TTA core. They are distinguished by the used memory type and the location of the memory controller.

single instruction for the fetch unit.

4.4 Stand-alone FPGA Integration

One of the use cases of the Platform Integrator is to integrate TTA processors into FPGAs as stand-alone processors. In this case, it is typical that the interfacing with the devices on the FPGA board is often implemented using Special Function Units (SFU) which contain device specific custom operations. The SFUs can perform rather simple operations such as reading button or switch states and lighting leds, or more complex operations such as configuring DAC chip parameters and playing audio samples. These SFUs are stored in platform specific HDBs which are used with the matching Platform Integrator.

Another characteristic aspect in the stand-alone FPGA integration is the mapping of the design signals to the FPGA pins to allow access to off-chip devices. Platform Integrator has the information on how to connect SFU signals to the correct pins, when the platform specific HDB is used. Naturally, the design clock signal also needs to be connected to a clock source and the reset signal is to be connected as well. The connection points of these signals are platform specific and possibly configurable. For example, there might be multiple clock sources of different frequency connected to the FPGA.

Synthesis is also characteristic of the stand-alone TTA integration. The stand-

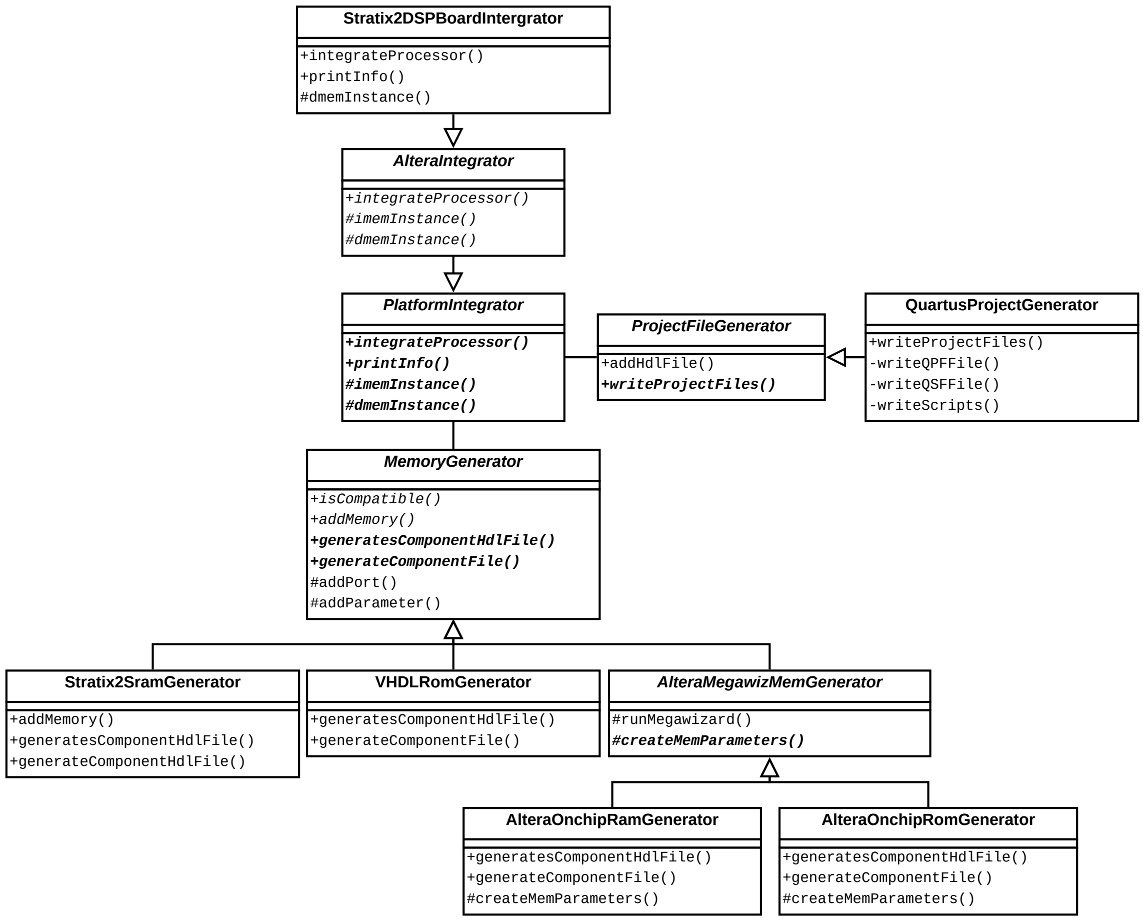


Figure 4.4: Class diagram of the Altera Stratix II DSP Board Integrator which integrates TTA cores as stand-alone processors on the FPGA evaluation board in question.

alone processor is prepared for synthesis, hence the project files for synthesis tools should be created for practicality. Furthermore, executable scripts for running the synthesis and programming the FPGA board could be written to simplify the FPGA flow.

A stand-alone FPGA integrator for Altera Stratix II DSP Board [35] was implemented. The class diagram of the Stratix II DSP Board Integrator (later Stratix II Integrator) is presented in Fig. 4.4.

On-chip memory components in the Altera platform can be created with a tool called *qmegawiz* [20]. A set of parameters is given to the tool which then generates a wrapper for the Altera component *altsyncram* with the user defined parameters. *AlteraMegawizMemGenerator* was implemented to call the *qmegawiz* tool in order to generate on-chip memory components during the execution of Platform Integrator. Two classes were derived from the *AlteraMegawizMemGenerator*: the first was *AlteraOnchipRamGenerator* for creating a RAM memory component and the other was *AlteraOnchipRomGenerator* which is used to create a read-only instruction memory component. Both of these classes provide the memory type specific parameter set needed by the *AlteraMegawizMemGenerator* to invoke the *qmegawiz*.

An off-chip memory generator *Stratix2SramGenerator* was also created for Stratix II Integrator. It is utilized to interface the TTA core with the SRAM memory device available on the FPGA evaluation board. The SRAM LSU, stored in the Stratix II specific HDB, was implemented in such way that the memory controller is included in the LSU. Thus, the *Stratix2SramGenerator* does not need to create a memory controller or a memory component file.

For the Altera specific integration generalizations, an abstract class *AlteraIntegrator* was derived from the *PlatformIntegrator*. The purpose of this class is to implement the *dmemInstance()* and *imemInstance()* methods to add support for Altera on-chip memory generators derived from the *AlteraMegawizMemGenerator*. This eases the use of on-chip memory components.

The Stratix II Integrator main class *Stratix2DSPBoardIntegrator* is derived from the *AlteraIntegrator*. The main class controls the integration process, uses *MemoryGenerators* to handle memory connections and handles FPGA pin mapping. Finally, it uses *QuartusProjectGenerator*, which is derived from *ProjectFileGenerator*, to create project files for the Altera Quartus II synthesis tool. These project files list, for example, the name of the design, FPGA device information, pin mapping information and all the HDL files needed to synthesize the design. *QuartusProjectGenerator* also writes shell scripts *quartus_synthesize.sh* and *quartus_program_fpga.sh* for easy FPGA synthesis and execution process, respectively.

4.5 SoC Design Flow Integration

The other main use case of the Platform Integrator is to wrap TTA cores to IP blocks which can be utilized in System-on-Chip designs. Characteristic in this use case is that the final system is constructed with 3rd party tools and synthesized afterwards. Therefore, the Platform Integrator does not need to perform FPGA pin mapping because it is done later on in the SoC design flow. However, synthesizing the TTA IP block generated by the Platform Integrator separately from the system is useful as it gives information about the resource usage and clock frequency of the TTA.

IP blocks communicate with each other via bus interfaces. In order to use TTAs in a SoC, different bus interfaces must be implemented for TTA. Bus interfaces can be interfaced from the software via SFUs which are used with custom operations or in case of memory mapped bus interfaces, the implementation can be embedded inside the load-store unit. These bus interface function units are stored to SoC platform specific hardware databases.

SoC design tools typically utilize IP block metadata to be able to identify the signals related to a certain bus interface. This is why the automatic metadata generation of a TTA IP is vital for fluently importing the generated TTA IPs to

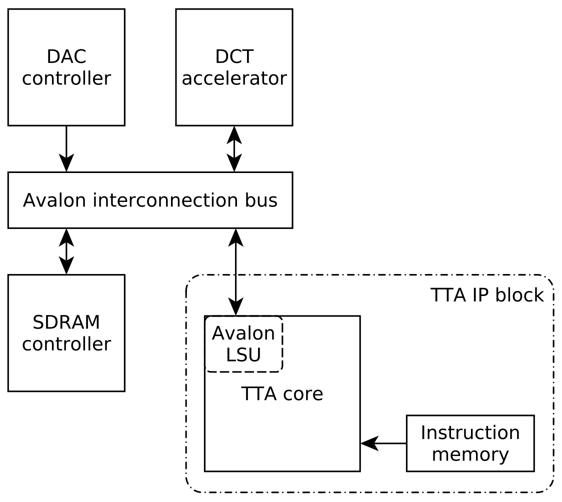


Figure 4.5: Example of a TTA IP block connected to the Avalon interconnection bus using the Avalon LSU. The data memory of the TTA resides in the SDRAM which is accessed using the SDRAM controller connected to Avalon.

other design tools. In the IP block integration, the Platform Integrator needs to map the external interface signals of a function unit to the logical bus signals for the metadata generation. This is similar to mapping signals to FPGA pins performed in the stand-alone FPGA integration.

4.5.1 Avalon Integrator

The first TTA IP integrator created for this thesis is the Avalon Integrator. The purpose of this integrator is to convert TTAs to Altera SOPC Builder compatible IP components. The final system can then be constructed with the SOPC Builder which allows the TTA to exploit other SOPC Builder compatible IP blocks.

In order to ensure the hardware compatibility of the TTA with the SOPC Builder, the Avalon Memory Mapped master interface was realized for TTA. Due to the memory mapped nature of the bus interface, it was implemented as a load-store unit. This Avalon LSU is not interfaced with a local data memory which means that the data memory of the processor, as well as the other IP blocks, are accessed via the Avalon interface. The type of the data memory is defined in the SOPC Builder and the FPGA board-specific memory controllers are then provided by the SOPC Builder. Therefore, the Avalon Integrator does not need to care about the data memory controllers if the Avalon LSU is used. An example of a system including an Avalon-compatible TTA is presented in Fig. 4.5. In this example, the TTA data memory is accessed using the SDRAM controller connected to the Avalon bus.

However, there are a few limitations set by TCE concerning SOPC Builder designs. First of all, TTA programs always assume that the data memory starts from the address zero due to the lack of a memory mapper in the current *tcecc* compiler.

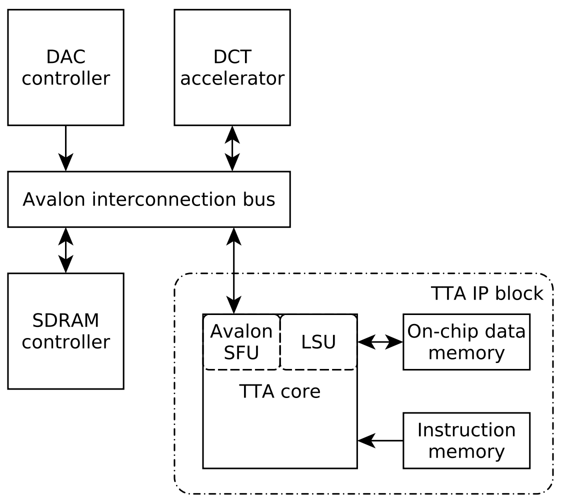


Figure 4.6: Example of a TTA IP connected to the Avalon interconnection bus using the Avalon SFU. A local memory LSU is used to access an on-chip data memory which is hidden from the rest of the system. The Avalon SFU is utilized to communicate with the other Avalon components.

In practice, this means that the base memory address of the data memory must be set to zero in the SOPC Builder. Fortunately, the IP component base addresses are user definable in the SOPC Builder so the issue can be easily circumvented. Another issue is the lack of interrupts in TTA. Avalon MM slaves can send interrupts to the master to notify about events. The Avalon LSU registers these interrupts but they do not interrupt the normal execution of TTA. The Avalon LSU provides a custom operation for reading the interrupt register and polling can be used to implement event handling.

The Avalon MM interface was also implemented as an SFU to allow TTA cores to have a local memory access which is independent of the Avalon bus congestion and hidden from other components in the system. This kind of setup is illustrated in Fig. 4.6. The SFU is basically the same as the Avalon LSU with the exception that the normal *load* and *store* operations are replaced by custom *avalon_load* and *avalon_store* operations. This is similar to using the *IOWR* and *IORD* macros in Nios II programs to access the Avalon components. Using these operations, the Avalon SFU can be used to access memories and other components connected to the Avalon bus like, for example, the SDRAM controller as shown in Fig. 4.6. When a local memory LSU is used with Avalon SFU, the Avalon Integrator must create and connect an on-chip memory component to the TTA core. Off-chip memory can also be used as an alternative. In this case, the memory interface signals are exported out of the SOPC Builder design using the Avalon conduit interface in the TTA IP component. The conduit interface can also be used for other ad-hoc connections.

The class diagram of the Avalon Integrator is illustrated in Fig. 4.7. It relies much

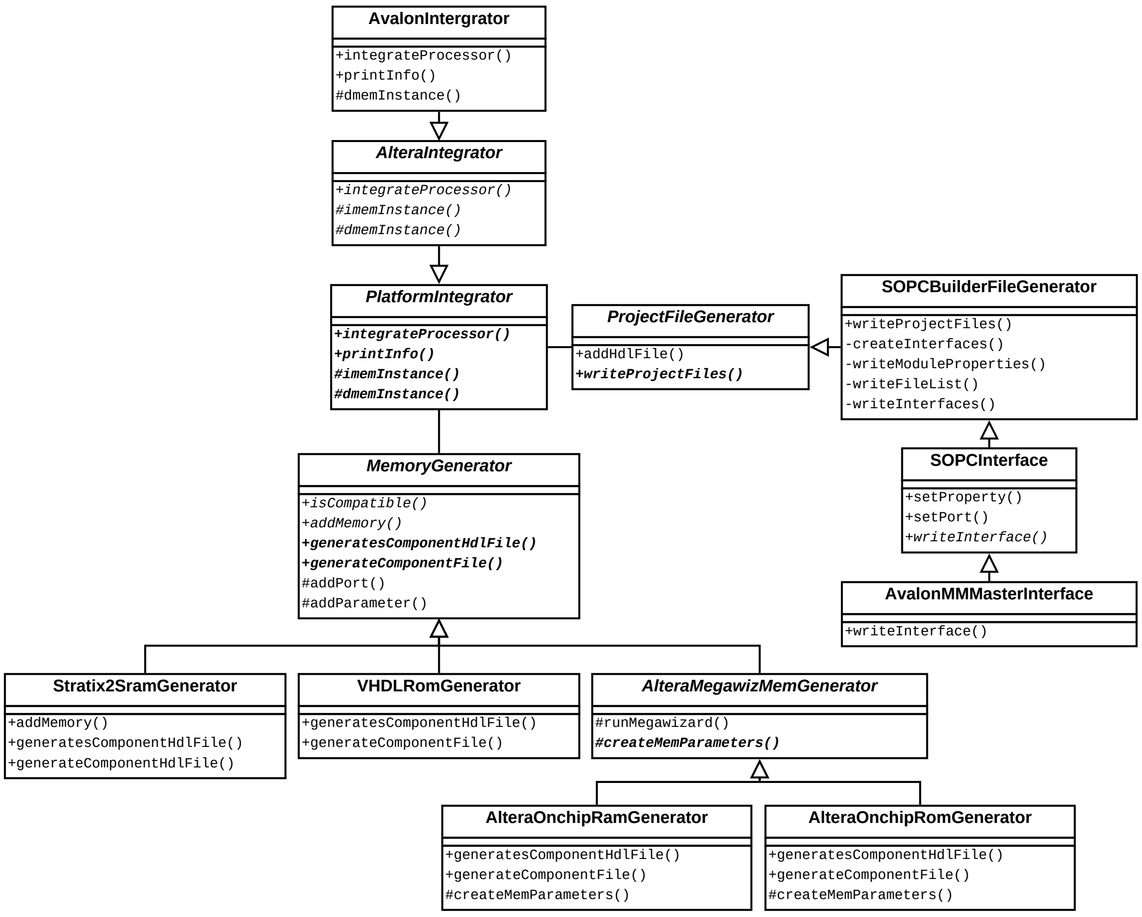


Figure 4.7: Class diagram of the the Avalon Integrator which can be used to wrap TTAs to Altera SOPC Builder compatible IP components

on the Altera specific framework created for the Stratix II Integrator. For example, there was no need to create new memory generators. *AvalonIntegrator* class was derived from *AlteraIntegrator* to control the integration process.

The metadata generation is handled in the *SOPCBuilderFileGenerator* class which searches the Avalon interfaces from the TTA core. The interfaces are matched according to the names, the directions and, if applicable, the widths of the interface signals. The *AvalonIntegrator* in synchronized with the Avalon specific HDB which guarantees successful integration. The interfaces are modeled using the *SOPCInterface* class from which *AvalonMMMasterInterface* is derived. These interface objects contain the signals associated to the bus interface as well as references to other associated interfaces. For example, every Avalon MM master is associated to an Avalon clock interface. Further Avalon interfaces could be supported by deriving new *SOPCInterfaces* and creating matching function units for them. Finally, the *SOPCBuilderFileGenerator* writes a *_hw.tcl* metadata file which is used to import the design into the SOPC Builder.

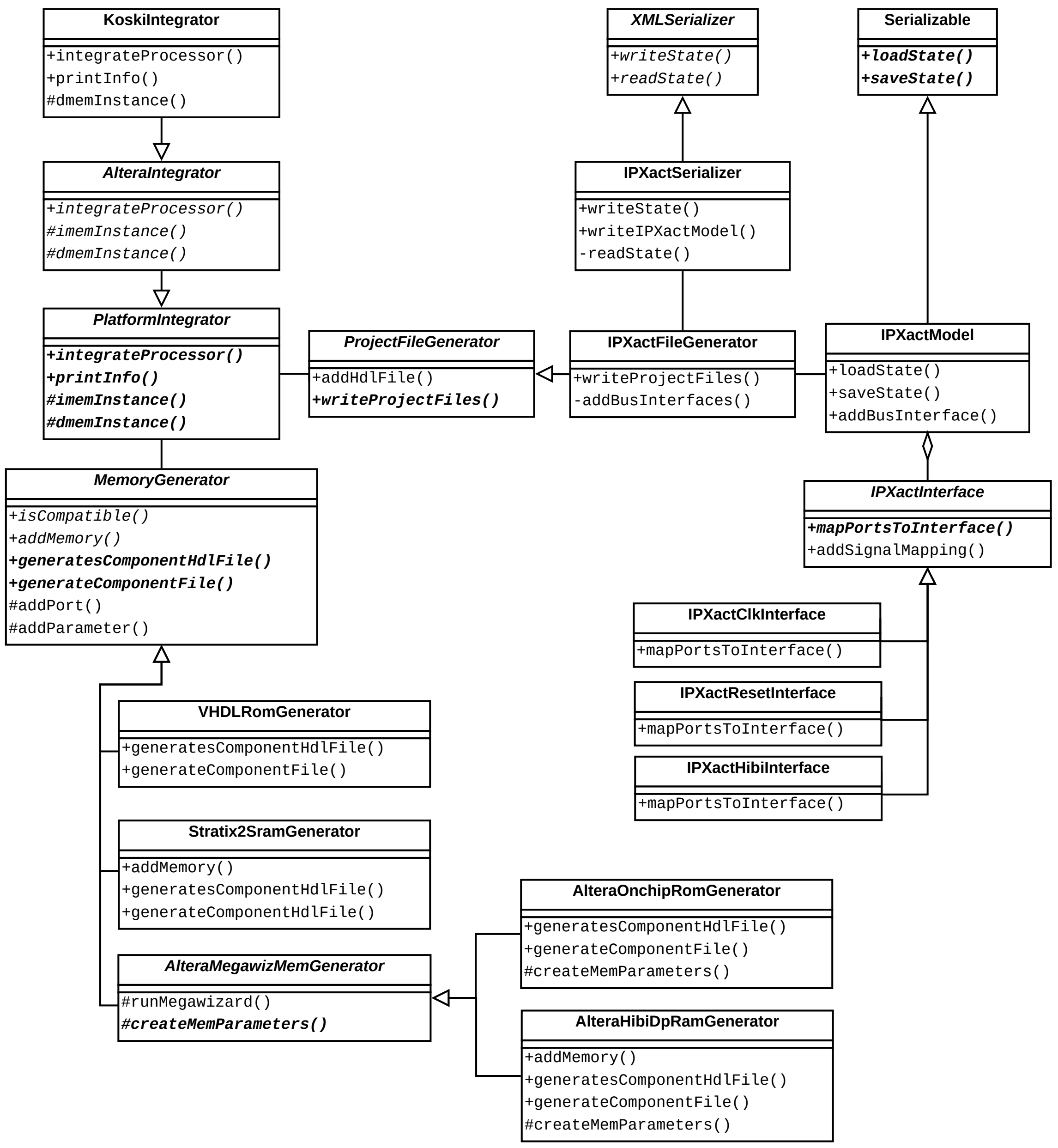


Figure 4.8: Class diagram of the Koski Integrator. The Koski Integrator is utilized to create Koski compatible TTA IP blocks.

4.5.2 Koski Integrator

Another IP integrator created for this thesis is the Koski Integrator which is used for creating Koski compatible TTA IP blocks. Hardware compatibility was achieved by implementing the HIBI interface for TTA. This implementation embedded the N2H2 IP block inside the LSU of TTA, and, thus, allowed DMA transfers via the HIBI bus. The HIBI LSU implementation is presented in more detail in [36].

The HIBI LSU requires dual ported RAM access for efficient DMA transfers. One port is reserved for the memory access of the TTA and the other one is for the N2H2. For this purpose, a new Altera on-chip memory generator called *AlteraHibiDpRam-*

Generator was derived from the *AlteraMegawizMemGenerator* as shown in the class diagram in Fig. 4.8. It works in a similar way as the other Altera memory generators and it recognizes the HIBI LSU memory interface. The Koski Integrator does not support off-chip data memory interfaces, unless they are implemented using ad-hoc connections. Even if an off-chip memory was used, the HIBI LSU would still require an on-chip dual port RAM for the DMA transfers.

The IP-XACT metadata generation is handled by the *IPXactFileGenerator*. This class creates an *IPXactModel* of the TTA which describes all of the TTA interface signals and HDL files. Furthermore, the TTA interface signals are mapped to appropriate *IPXactBusInterfaces* when applicable. For example, the HIBI interface in the HIBI LSU is recognized and modeled using the *IPXactHibiInterface*. The IP-XACT interface support can be extended by implementing new *IPXactInterface* classes and by creating matching function units to provide the hardware level compatibility.

The IP-XACT metadata is stored in the XML format. In order to easily write the IP-XACT files, the *IPXactModel* class was derived from the *Serializable* class of TCE base library which enforces the child class to implement the *loadState()* and *saveState()* methods. The *IPXactSerializer* class, derived from the *XMLSerializer* of TCE base library, uses these methods to write and read IP-XACT files. TCE version 1.3 supported IP-XACT version 1.2. IP-XACT support was updated to version 1.5 before TCE 1.4 was released.

5. VERIFICATION FLOW

Design verification is at least as important as the implementation. Everything that is designed must be tested and verified because otherwise there is no certainty that the design behaves as specified. Verification can take half of the time of the whole design process, if not more. [37]

This chapter describes the verification methods of TTA processors designed with TCE tools. First, the verification flow of TTA processors is discussed. Then, a new TTA Unit Tester verification tool for unit testing processor datapath resources is introduced.

5.1 Top-down Verification

Verification can be done on multiple abstraction levels of the design process. The top-down approach allows the possible faults to be found on the highest possible abstraction level which helps to isolate and target the source of the fault. TCE verification levels and methods are presented in Fig. 5.1. In the following sections, the different verification levels are discussed in more detail followed by the descriptions of the two different verification methods shown in Fig. 5.1.

5.1.1 Verification Levels

The top-down verification starts from the highest possible abstraction level and moves to lower abstractions in steps. After each level transition, it must be ensured that the design behavior does not change before moving on in the flow. One of the key motivation factors in this approach is that typically the design debugging becomes harder and harder as the design abstraction moves lower towards the real implementation. At the same time, the verification execution time increases with the exception of the FPGA execution. The levels of the TCE top-down verification flow are the following.

Desktop Execution

First step of the verification is to compile and execute the HLL application on a standard desktop PC. A wide range of software development and debugging tools are available in the desktop PC environment making it the most suitable option for verifying the application software and generating test data for the later levels.

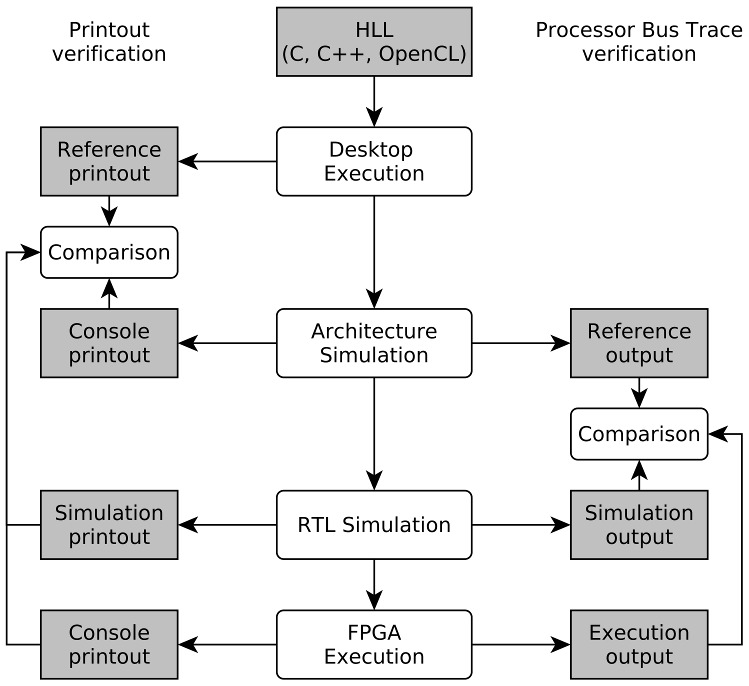


Figure 5.1: Top-down verification of TTA processors. The verification method based on printouts is presented on the left side and on the right is the processor transport bus trace based method. Both methods can be applied on various levels of the design process.

In general, the application execution speed is fast since desktop processor clock frequencies are in the order of gigahertz.

Architecture Simulation

The second verification level, architecture simulation, is the first one involving a TTA processor. The architecture simulation is executed using the retargetable instruction-set simulator of TCE, either *ttasim* or *Proxim*. First of all, this level verifies the program portability to the TTA environment. If the program behavior changes compared to the desktop PC execution, it might indicate that the program source code does not comply with the programming language standards or it uses nonportable code. In rare cases, the culprit might be the compiler or the ISS.

The most of the design work in TCE is done at the architecture level which makes this an important verification level. Every time the processor architecture is changed, it must be confirmed that the program behavior remains unchanged. In order to allow fast design space exploration, the verification should be as automatic as possible.

The ISS of TCE provides versatile software debugging capabilities. The user can single step instructions, assign software breakpoints, examine the contents of register files and memories et cetera. Despite the program execution speed on the instruction-set simulator is significantly lower than on a desktop PC, often the performance, measured in simulated instruction cycles per second, is still acceptable. The overall simulation clock frequency depends on the TTA processor architecture and the program complexity, but often the frequency is in the megahertz range. [38] [39]

RTL Simulation

The next verification level is the RTL simulation which can be executed once the processor HDL implementation has been generated with ProGe. Before proceeding to simulate the whole processor, it is advisable to run the TTA Unit Tester (described in Section 5.2) to ensure the processor units are not defective. From this perspective, simulating the whole processor is in fact integration testing. Assuming that the units are verified to work, the errors found on this level are likely to be caused by integration issues.

The fundamental difference in the architecture and RTL simulation is that the RTL simulation is focused on observing and debugging the hardware operation rather than the software. While the user can inspect memory and register file contents as well as the internal registers of the function units, software breakpoints are not available per se. In other words, the RTL simulation gives a detailed view of the hardware execution but has very limited means to debug the software. The greatest disadvantage of the RTL simulation is the low simulation clock frequency which can be, depending on the design size and complexity, up to six orders of magnitude slower than the real hardware execution [40] and often two orders of magnitude slower than the architecture simulation.

FPGA Execution

The final verification level is executing the design on FPGA hardware. If the RTL simulation was a success, this level verifies that the integration to the target FPGA and the synthesis process were successful. For example, mapping the design signals to the FPGA pins can be surprisingly error prone if done manually. Fortunately, the Platform Integrator does this automatically. One possible pitfall after the synthesis is executing a design which does not meet the timing constraints. Some synthesis tools, like the Altera Quartus II, will write the FPGA programming file regardless of the timing analyser result, making it easy to accidentally test such configuration on the target hardware.

The FPGA execution breaks the trend of the execution speed slowdown in the verification because the design can be executed on high clock frequency. On the downside, the visibility to the hardware state is mostly lost. That is to say, the FPGA execution can be done quickly but usually the verification output is binary:

the design either works or does not work. In-system logic analyzers can be used to redeem the situation to some extent. They provide capabilities to capture signal waveforms during execution but the captured data size is usually quite limited.

If the design is to be implemented as an ASIC, the FPGA execution would be just another intermediate level before the final product. Typically, an FPGA might not be able to reach the high target clock frequency of an ASIC but the verification speed on the FPGA is significantly faster in comparison to the RTL simulation. The advantage of the FPGA verification is that the design can be tested on hardware before the final target hardware is manufactured.

5.1.2 Printouts

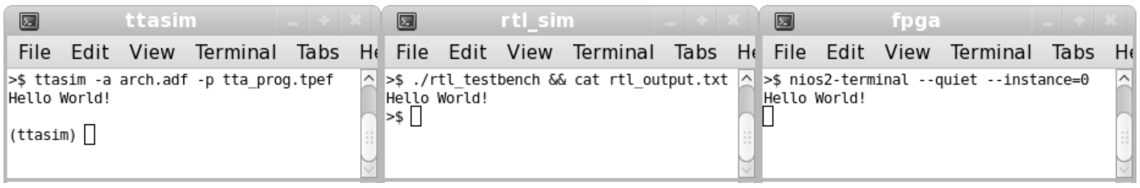
The printout verification method presented on the left side of Fig. 5.1 relies on printing data during the execution. This acquired data is used to verify calculation results and track the program execution. The standard C functions such as *printf()*, *puts()* and *putchar()* can be utilized for this purpose which ensures the verification code portability between the desktop PC and the TTA environment. Thus, the reference printout is created on the desktop PC environment and the printouts from the subsequent levels are compared to this reference.

Printing in the TTA environment requires the processor architecture to include a function unit which implements an operation called *STDOUT*. This operation is utilized to output a single character to a "standard output device", whatever that is in the integrated platform. In TCE, the standard C printing functions are implemented in such a way that they utilize the *STDOUT* operation. By default, the simulation model of this operation outputs the printed characters to the main window of the instruction-set simulator.

The RTL simulation requires an implementation for the function unit containing the *STDOUT* operation. Fortunately, the RTL simulation implementation does not have to be synthesizable, thus, for example, the *textio*-package of VHDL can be used to print the characters to a text file. However, for the FPGA execution the implementation must be synthesizable and the implementation method is target FPGA board dependent. Typically, FPGA evaluation boards provide an *UART (Universal Asynchronous Receiver Transmitter)* connection via a serial cable or a *JTAG (Joint Test Action Group)* cable to communicate with the host PC. FPGA boards might also have a display or a display connector which allows the characters to be printed on a separate screen but this approach would prevent the automatic printout comparison.

Figure 5.2 shows an example of acquiring the TTA printouts from different the verification levels. The TTA executes a trivial hello world application:

```
#include <stdio.h>
int main() {
```



```
>$ ttasim -a arch.adf -p tta_prog.tpef
Hello World!

(ttasim)
```

```
>$ ./rtl_testbench && cat rtl_output.txt
Hello World!
>$
```

```
>$ nios2-terminal --quiet --instance=0
Hello World!
```

Figure 5.2: Example of acquiring printouts from the TTA. On the left is a screen capture from the *ttasim* instruction-set simulator. The middle terminal executes the RTL simulation with *GHDL* and shows the generated printout file. On the right side window, the *nios2-terminal* program is executed to the capture characters sent from the FPGA evaluation board.

```
    printf("Hello World!\n");
    return 0;
}
```

which outputs the line "`Hello World!`". The left window in the Fig. 5.2 shows the output from *ttasim*, and the RTL simulation with *GHDL* is executed in the middle window. On the right is the output from the Altera Stratix II FPGA evaluation board. The printing is implemented using the *JTAG UART* IP block provided by Altera which allows the characters to be captured in the *nios2-terminal* program executed on the host PC.

5.1.3 Bus Trace

The bus trace verification method is based on recording the transport bus values of the processor at each clock cycle. This allows the execution to be verified on a much greater resolution than with the printout method. However, the bus traces cannot be acquired from the desktop PC execution because the transport buses are TTA specific. Thus, the bus trace cannot be used to verify the program porting between the desktop PC and TTA environment.

The instruction-set simulator of TCE saves the bus trace to a text file when the bus trace option is enabled. Bus traces can also be obtained from the RTL simulation if ProGe is instructed to generate a bus tracing module to the interconnection network of the processor. This module is implemented using the *textio*-package of VHDL, thus, it cannot be synthesized. The module prints the trace to a text file for easy comparison. Bus traces could also be recorded from the FPGA execution, for example, using a JTAG based bus tracing module. However, such module has not been implemented yet.

The drawback of using the bus trace method is that it slows down the simulation. This is due to the increased filesystem I/O operations caused by the bus trace data saving. The bus trace acquisition will probably slow down the FPGA execution as well if the bus trace data cannot be transferred unintrusively. In other words, the processor might need to be stalled during the time the data is transferred to the host PC.

5.2 TTA Unit Tester

As most of the design work in TCE is done on the architecture level, it is important to verify that the simulation models of processor datapath components, function units and register files, behave equally to their respective RTL implementations stored in HDBs. Otherwise, the processor implementation generated by ProGe may differ from its architecture model and behave differently. Testing the function unit implementations separately before the RTL simulation of the whole processor also helps in isolating the sources of the potential faults.

In order to ease the FU and RF testing, a tool called TTA Unit Tester was created. The TTA Unit Tester tool should be highly automated and should not require user intervention in testing. The tool needs to:

- be able to fetch the FU and RF implementations from the HDBs specified in the IDF.
- create randomized input stimuli for the tested units and generate the reference output using the operation simulation models.
- automate the 3rd party RTL simulator execution
- have the ability to save the created testbench files and output the RTL simulation commands for unit debugging purposes.

The testing framework should also be constructed in such a way that it provides a general-purpose interface for testing arbitrary units stored in an HDB. This allows the framework also to be used for testing hardware databases and possibly be integrated into the *HDBManager* tool.

The operation principle of the TTA Unit Tester tool is illustrated in Fig. 5.3. The function units and register files (later units) to be tested are identified by the processor implementation definition file given to the tool. One by one, the tool will fetch a unit implementation from an HDB and recognizes the operations which the unit implements. The TTA Unit Tester will then generate input stimuli vectors for all the operations implemented by the unit. Operation reference output vectors are created by using the corresponding operation simulation models. Next, the tool creates an RTL testbench for the unit under test. The testbench feeds the input stimuli to the unit and compares the produced output with the reference output vectors. If they differ, the testbench produces failing assertions.

In order to simulate the testbench, an RTL simulator is needed. For this purpose, the TTA Unit Tester uses a 3rd party RTL simulator, such as the open source

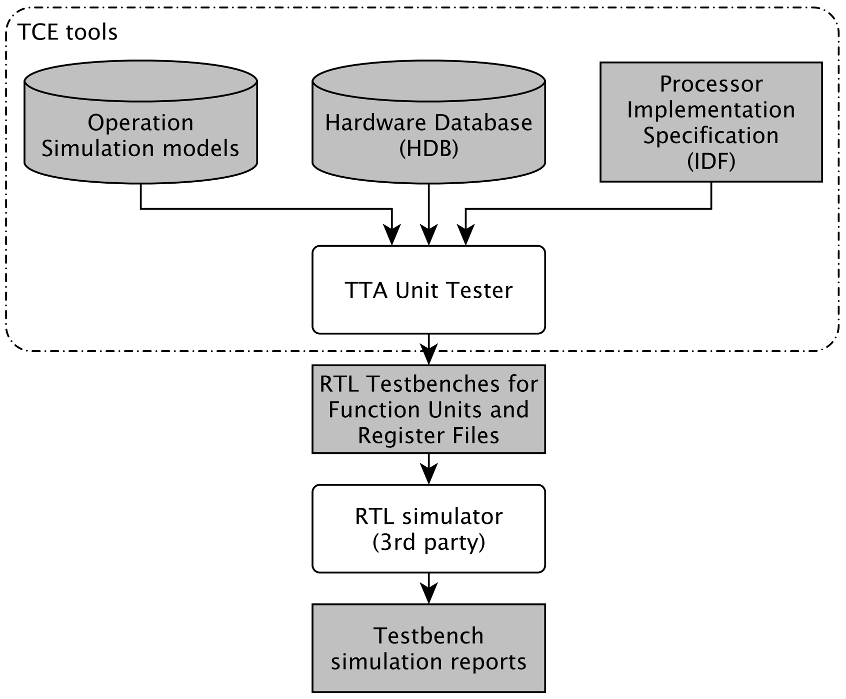

TCE tools
Operation Simulation models
Hardware Database (HDB)
Processor Implementation Specification (IDF)
TTA Unit Tester
RTL Testbenches for Function Units and Register Files
RTL simulator (3rd party)
Testbench simulation reports

Figure 5.3: The operation principle of the TTA Unit Tester. The TTA Unit Tester creates input stimuli for the tested operations and uses the operation simulation models to create the reference output. Using this data, an RTL testbench is generated for the unit implementation which is executed using a 3rd party RTL simulator.

GHDL [41] or the commercial ModelSim [42], to execute the testbench simulation. The simulator outputs testbench simulation reports which are used to determine whether the test was successful or not.

The TTA Unit Tester can also be useful when creating function unit implementations for custom operations. Typically, the function unit simulation model is available when custom operations are implemented which allows the TTA Unit Tester to be utilized during the implementation process. The custom operation implementation is iterated until it passes the TTA Unit Tester.

It should be noted that this kind of method cannot be used to provide a thorough verification of nontrivial units because the number of possible input data combination increases exponentially as a function of input signals. For example, a unit with a single 32-bit input port has 2^{32} possible input combinations alone and adding the function unit control signals, such as the input operand load signal and the global lock signal, to the equation increases the number of possible combinations even further. In case the testbench gives failing assertions, it can be determined that the architecture model and the implementation of a unit differ from each other. If the testbench execution is successful, it only verifies that the unit architecture model and implementation are equal on some input combinations but not necessarily on all input combinations.

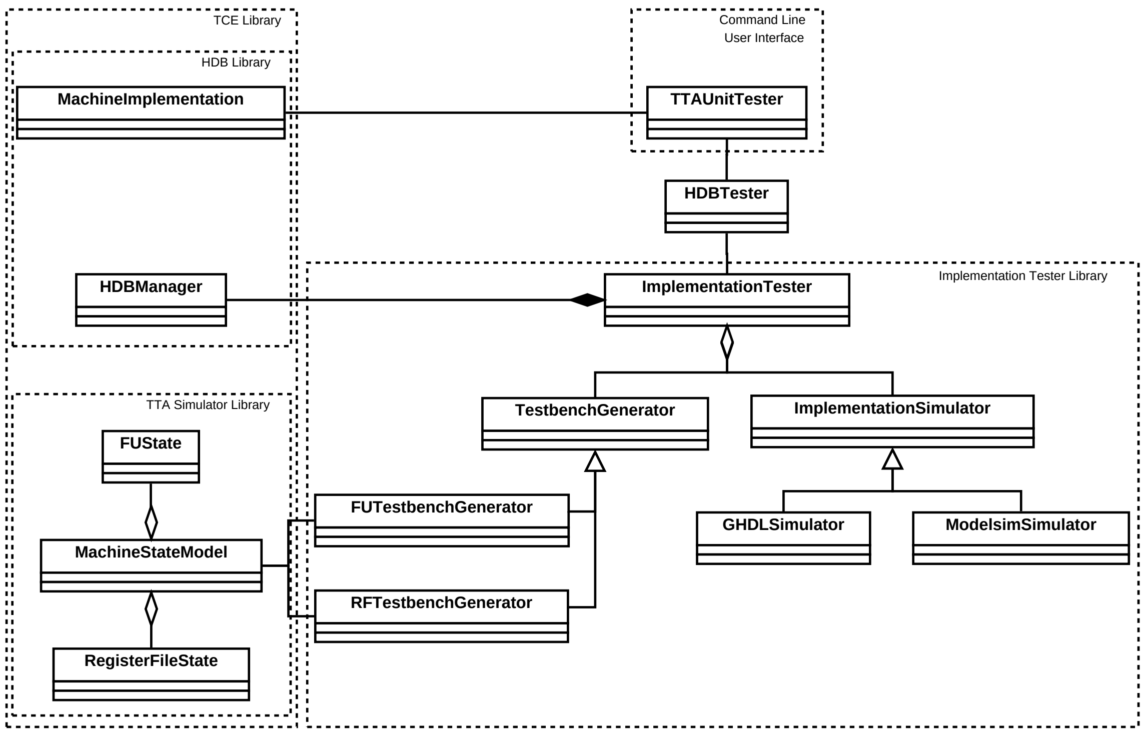


Figure 5.4: Class diagram of TTA Unit Tester. Implementation was divided to an User Interface component and a separate Implementation Tester Library component. Tool also uses existing library components from TCE Library for accessing HDBs and simulating the tested units.

The class diagram of the TTA Unit Tester is presented in Fig. 5.4. The *TTAUnitTester* module provides a command line user interface to the tool. *TTAUnitTester* takes the IDF of the processor to be tested as a parameter and constructs a *MachineImplementation* object model out of it. The *MachineImplementation* is used to identify in which HDB files the tested units are stored. The *HDBTester* is a middle class which generalizes the testing of units stored in a HDB by creating an *ImplementationTester* object for each requested HDB. It also provides methods for testing a single FU or RF entry or all entries inside the HDB.

The actual functionality resides in the Implementation Tester Library. The *ImplementationTester* is the main class of the library. It uses the *HDBManager* to fetch FU and RF entries from HDBs and then verifies that the specified FU or RF can be tested. An FU cannot be tested if:

- the HDB entry of the FU lacks architecture or implementation, because in this case the unit cannot be used in both architecture simulation and RTL simulation,
- the FU is not fully pipelined. However, support for complex pipelines could be added later on,
- the FU connects to a memory because the HDB does not contain detailed

information on the expected memory component,

- the FU has external ports because the behavior of the unit might depend on external events or
- the FU has only one unidirectional port because in this case, the input stimulus cannot be written to the unit or the effect of input stimulus cannot be observed.

In case the unit cannot be tested, appropriate messages are displayed to the user.

The *TestbenchGenerator* class generalizes the RTL testbench creation. The classes *FUTestbenchGenerator* and *RFTestbenchGenerator*, derived from the *Testbench-Generator*, handle the unit type specific sections through the virtual function *generateTestbench()*. They construct a *MachineStateModel* which includes the unit under test. The *MachineStateModel* implements the architecture simulation of the unit which is used to create the reference output from the generated input stimuli. At the end, the RTL testbench is written to a specific file.

Interfacing with the RTL simulation tools is implemented in the *ImplementationSimulator* class. This abstract base class has methods *compile()* and *simulate()* which are realized in the derived simulation tool specific classes *GHDLSimulator* and *ModelsimSimulator*. The subclass executes the simulation and filters the output messages for testbench assertions and simulation tool specific error messages, which are displayed to the user if found.

6. EVALUATION OF RESULTS

This chapter describes how the Platform Integrators implemented in this thesis are verified to work using a test application. This test application is implemented using a TTA design on all three platforms and the results are verified according to the verification flow described in Section 5. The following section briefly introduces the test application and the initial modifications for verification purposes. After that, the ASIP design process for this application is summarized followed by the platform specific sections. All the test phases in the following sections were executed on a VirtualBox [43] version 3.2.4 virtual machine running the 32-bit Fedora 14 Linux operating system. The host PC is equipped with an Intel Core 2 Quad Q9400 processor running in 2.66 GHz clock frequency and 4 GB of RAM and it runs the Windows XP SP3 operating system.

6.1 Test Application

Cyclic Redundancy Check (CRC) is a checksum algorithm designed to be used as an error-detection method in data communications and storage. There is a variety of different polynomials which can be used for CRC, but in this case, the CRC-32-IEEE 802.3 was chosen because it is commonly used in Ethernet [44] and some other standards. The CRC-32 evaluates a 32-bit checksum from a variable length of input data.

The implementation of the CRC-32 used in this thesis is written by Michael Barr and the C language source code is released under a public domain license [45]. The code includes three different polynomials and two methods for calculating CRC, but only the CRC-32-IEEE 802.3 polynomial and the faster method for calculating CRC are considered in this case.

The first step in the design process was to compile the code for the desktop PC and verify that the program works. Next, random input data sets were created and the corresponding checksums were evaluated and stored for verification purposes. The data set sizes range from 60 to 1518 bytes which is the size of an Ethernet frame without the checksum [44]. These data sets were used to verify the TTA implementation. An automatic result verification system was also created using the reference checksums stored in the data sets. If the evaluated checksum equals to the reference, letter *O* is printed and in case they differ, letter *N* is printed. The CRC calculation was set to be iterated 100 times to be able to measure the run times more accurately since calculating a single CRC does not take a very long time. Finally, the reference output of the desktop PC execution was stored to a text file for verification purposes according to the Fig. 5.1. For later comparison, the execution time of the test application was measured on the desktop PC using the *gettimeofday()* function. As expected, the execution time on the desktop PC was short, around two milliseconds.

6.2 Customized Processor for CRC

The first step in the ASIP design flow presented in Fig. 3.2 is to choose a starting point architecture and then compile and simulate the application. From the point of view of the verification flow, the application portability to TTA is verified on this step. A minimalistic TTA, which contains the minimum resources needed by the *tcecc* to compile C code, was used as the starting point. However, *Real Time Clock (RTC)* and STDOUT function units where added to the architecture to allow the accurate measuring of the execution time and printing from the TTA. After the architecture simulation with *ttasim*, the simulation console printout was verified to be equal with the reference printout from the desktop PC execution.

Unsurprisingly, the minimalistic TTA did not offer optimal performance. The execution time for iterating 100 times the calculation of the CRC checksum from 1 518 bytes of input data was 520 582 ms assuming a clock frequency of 100 MHz. The performance of the architecture can be measured in *megabits per second (Mbps)*, which tells how much input data can be processed in one second. The minimalistic TTA can merely calculate CRC checksums at the rate of 2.22 Mbps. Therefore, the iterative processor design space exploration was started to increase the performance. The number of transport buses and registers was increased and the function unit setup was optimized. [36]

Execution profiling revealed that the most of the execution time was spent performing bit pattern reflections. In software, the bit pattern is reflected iteratively one bit at a time. However, on hardware, the bit pattern reflection can be implemented with simple crosswiring which allows the whole bit pattern to be reflected at once. The bit pattern reflections for 8 and 32 bit data widths were implemented as custom operations which resulted in a notable speed up: the execution time dropped from 82.38 ms to 21.27 ms. Measured in throughput, the performance increased from 14.06 Mbps to 54.45 Mbps. The final architecture is called CRC TTA and the resources of the architecture are listed in Table 6.1.

The architecture simulation time of the CRC TTA executing the CRC application is short, around 1.5 seconds, measured with the Linux command line program *time*. Overall simulation cycle count, including the result verification and the calculation time printing, is 2 132 934 cycles. Thus, in this case, the simulator executes around 1.4 million instructions cycles per second which equals to a simulation frequency of

Table 6.1: Datapath resources of the CRC TTA processor

Resource name	#	Description
Alu_comp	1	FU with operations: add, sub, eq, gt, gtu
Logic	1	FU with operations: and, ior, xor
Shifter	1	FU with operations: shl, shr, shru
Reflecter	1	FU with custom operations: reflect8 ja reflect32
LSU	1	Load-store unit
RTC	1	FU with operations: RTC, RTIMER
IO	1	FU which implements STDOUT operation
Register file	1	Includes 16 32-bit registers, 1 read and 1 write port
Boolean RF	1	Includes 2 1-bit registers, 1 read and 1 write port
IU	1	Long immediate unit with 1 32-bit registers and 1 read port
Control Unit	1	Control Unit of the processor
Transport bus	3	Fully connected transport bus

1.4 MHz.

6.3 Stratix II Integrator

The Stratix II Integrator was used to integrate the designed CRC TTA for stand-alone execution on the Stratix II FPGA evaluation board. The main purpose of this section is to verify the operation of the Stratix II Integrator. For performance comparison to the existing soft-cores, the same CRC application was implemented on the Altera Nios II/f soft-core processor to give a reference point for the performance.

The CRC TTA processor was integrated into the Stratix II using on-chip memories for instruction and data memory. In this case, the stand-alone integration did not require any modifications to the architecture. The implementations for the platform specific function units, such as the load-store unit and the *STDOUT* function unit, must be selected from the Stratix II HDB for successful stand-alone integration. At this point, the selected *STDOUT* implementation used nonsynthesizable structures for RTL simulation purposes. Before moving to the RTL simulation, the TTA Unit Tester was utilized to verify the function units and register files.

The RTL simulation is the third verification level in the verification flow illustrated in Fig. 5.1. The RTL simulation was executed with Modelsim 6.6d using its default settings. Both the printout and bus trace methods were used to verify the behavior of the processor. The execution time of the RTL simulation was 2 minutes 55 seconds which equal 175 seconds. The simulation clock cycle count was 5 cycles longer compared to the architecture simulation due to initializations. At the beginning of simulation, the reset signal is active and the first instruction is executed after the instruction fetch delay when the reset signal is released. The total simulation

Table 6.2: Performance and logic element (LE) utilization comparison between the different TTA processors and Nios II/f implementations with and without custom operations. The clock frequency was 100 MHz on all of the test cases.

Architecture	**Execution time / ms**	**Performance / Mbps**	**LE Usage**
Minimalistic TTA	520.58	2.22	1 248
CRC TTA w/o custom op	82.38	14.06	1 738
CRC TTA w/ custom op	21.27	54.45	1 810
Avalon LSU TTA	25.52	44.82	1 911
Avalon SFU TTA	21.27	54.45	1 990
Nios II/f	215.66	5.63	1 563
Nios II/f w/ custom op	37.98	31.98	1 563

cycle count is then 2 132 939 cycles. Thus, the RTL simulator executed 12 188 clock cycles per second which equals to a simulation clock frequency of approximately 12.2 kHz. In this case, the RTL simulation was two orders of magnitude slower compared to the architecture simulation.

The final step was to synthesize the design for the target FPGA. For synthesis, the *STDOUT* implementation was switched to a synthesizable JTAG UART based one, and the processor generation and integration was executed again. The synthesis was performed with Quartus II version 8.0 and the results are listed in Table 6.2. The execution on FPGA was fast since the target clock frequency of 100 MHz was used, and, thus, the execution time was approximately 21.27 milliseconds. However, the dominant time in the FPGA verification phase, apart from the synthesis, is the time consumed in FPGA programming which took 16.2 seconds in this case.

The same CRC application was implemented on Nios II/f to provide a reference point for the performance of the TTA designs. The block diagram of the Nios II system constructed with the SOPC Builder is presented in Fig. 6.1 which shows the essential components needed for testing. On-chip memories were used to eliminate the differences caused by the memory latency. The same reflection custom operations were also implemented for Nios II as an instruction set extension and the application was executed with and without the custom operations. The target clock frequency was set to the same 100 MHz. [36]

The synthesis and performance results of the Nios II and the CRC TTA are listed in Table 6.2. Both of the Nios II test cases were executed on the same hardware, thus the logic element usage is constant. The results show that the standard Nios II/f lacks the ability to exploit ILP as well as a TTA and therefore the performance is inferior. Extending the Nios II/f instruction-set with the reflection operations improved the performance because the most time consuming part of the algorithm was executed on hardware. But the performance was still only about 59 % of the

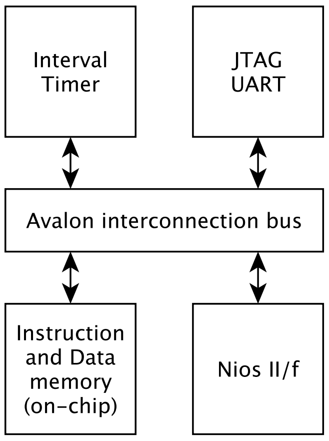

Interval Timer
JTAG UART
Avalon interconnection bus
Instruction and Data memory (on-chip)
Nios II/f

Figure 6.1: The Nios II/f system used in the performance comparison. The interval timer IP block is used to accurately measure the application execution time and the JTAG UART was utilized to print results from the execution.

performance of the CRC TTA with the custom operations. In addition, the CRC TTA with the custom operations only consumes approximately 17 % more logic elements than the Nios II/f which is justified considering the performance boost.

6.4 Avalon Integrator

The verification of the Avalon Integrator is performed in this section using the same CRC test application. This section is divided in two parts. First, the Avalon Integrator is tested using the Avalon LSU and then with the Avalon SFU. In order to demonstrate the ability of TTA to utilize SOPC Builder IP blocks, changes were made to the ASIP architecture which was created in Section 6.2. Both the RTC and STDOUT function units were removed from the architecture and they were replaced with the interval timer and JTAG UART Avalon IP components [46].

In order to use the Avalon Integrator, an Avalon function unit must be included in the processor architecture. First, the local memory load-store unit is replaced with the Avalon LSU. Because the Avalon LSU has one additional operation, *AVALON_READ_IRQ*, compared to the local memory LSU, the switch cannot be done simply by defining a different implementation in the IDF before generating the processor.

Because the interval timer and the JTAG UART are now SOPC Builder IP blocks, the application source code must also be updated. The interval timer [46] must be initialized to act as a high resolution timer by writing to its memory mapped configuration register. Timer snapshots are acquired by first writing to the memory mapped snapshot register which triggers the IP block to save the current timer value. Then the value can be read from the snapshot register. In order to enable printing with the Avalon JTAG UART [46], a new *putchar()* implementation, described in

```
int putchar(int ch) {
  volatile int* ctrl = (int*) JTAG_UART_CTRL_REG_ADDR;
  volatile int* data = (int*) JTAG_UART_DATA_REG_ADDR;
  /* wait until space in buffer */
  while ((*ctrl & JTAG_UART_WRITE_BUFFER_MASK) == 0)
    ;
  *dataReg = (int) ch;
  return ch;
}
```

Figure 6.2: Implementing *putchar()* using JTAG UART IP block via the Avalon LSU

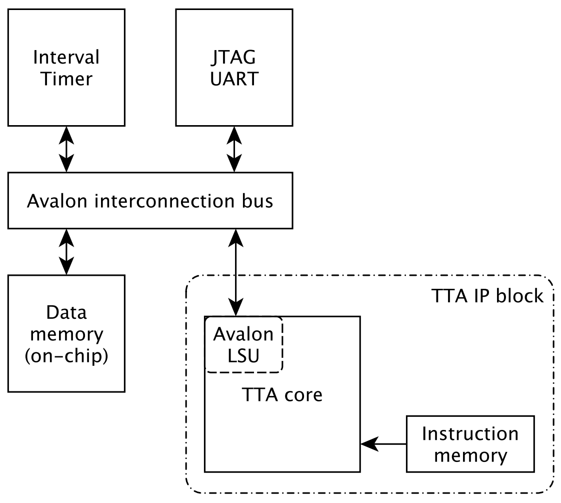


Figure 6.3: A SoPC design with a TTA IP. The TTA core uses the Avalon load-store unit (LSU) to interface with the Avalon bus. The data memory of the TTA, like the other IP blocks, are accessed via Avalon.

Fig. 6.2, is defined. This function first waits until there is space in the write data buffer and then writes the character to the data buffer. As the IP block registers are memory mapped, they can be accessed using pointers.

The downside of using the Avalon IP blocks is that they make the verification process more complicated. These IP blocks are not usable in the architecture simulation, and in this case, it means that the printouts cannot be acquired from *ttasim*. One way to circumvent this issue is to include the STDOUT function unit in the architecture during architecture simulation to acquire the printouts and then remove this function unit before proceeding to the next verification level. The utilization of the IP blocks also complicates the bus trace verification if the execution flow of the application depends on external events. For example, if a memory mapped register is polled while waiting for a specific value, the number of polling iterations may vary resulting in differing bus traces. Thus, the verification of a TTA IP requires more effort of the designer.

When the architecture and source code modifications are ready, the Avalon Integrator is executed to create a SOPC Builder IP component of the TTA. Then, the TTA IP can be imported to the SOPC Builder by adding the TTA directory to the IP search path of the SOPC Builder. The block diagram of the created TTA SOPC system is presented in Fig. 6.3. The system was simulated with Modelsim to verify correct operation. Calculating the 100 iterations of the CRC with the Avalon LSU TTA system took 25.52 ms which equals a throughput of 44.82 Mbps. The performance reduction is due to the longer memory access latency via Avalon compared to a local memory access.

The last step was to synthesize the system and verity its operation on the FPGA. The synthesis results are listed in Table 6.2. The printout acquired from the FPGA execution verified that the system produced correct results.

As another test case, the Avalon bus is accessed using the Avalon SFU, which allows a local memory LSU to be used for accessing the data memory. Therefore, the architecture needs to include both a local memory LSU and the Avalon SFU. This setup required changes to the IP block controlling functions. The memory mapped registers could no longer be accessed using pointers since the LSU was no longer interfaced with the Avalon bus. Instead, *_TCE_AVALON_LDW* and *_TCE_AVALON_STW* custom operation macros are utilized to read and write 32-bit words via the Avalon. For example, the modifications needed for the *putchar()* implementation presented in Fig. 6.2 are illustrated in Fig. 6.4.

The TTA IP with Avalon SFU was created with the Avalon Integrator. In this setup, the Avalon Integrator also created the data memory component for the TTA IP as illustrated in Fig. 6.5. The created system was simulated with Modelsim to verify the operation and then synthesized. The synthesis results are listed in Table 6.2 which shows that the current system is 79 LEs bigger than the Avalon LSU TTA system. The difference is explained by the need for both the Avalon SFU and the local memory LSU instead of just the Avalon LSU. Finally, the FPGA execution verified that the system functioned correctly also after the synthesis. The results show that the CRC calculation time has dropped to 21.27 ms, thanks to the

```
int putchar(int ch) {
  unsigned int reg = 0;
  /* wait until space in buffer */
  while ((reg & JTAG_UART_WRITE_BUFFER_MASK) == 0) {
    _TCE_AVALON_LDW(JTAG_UART_CTRL_REG_ADDR, reg);
  }
  _TCE_AVALON_STW(JTAG_UART_DATA_REG_ADDR, ch);
  return ch;
}
```

Figure 6.4: Implementing *putchar()* using the JTAG UART IP block via the Avalon SFU

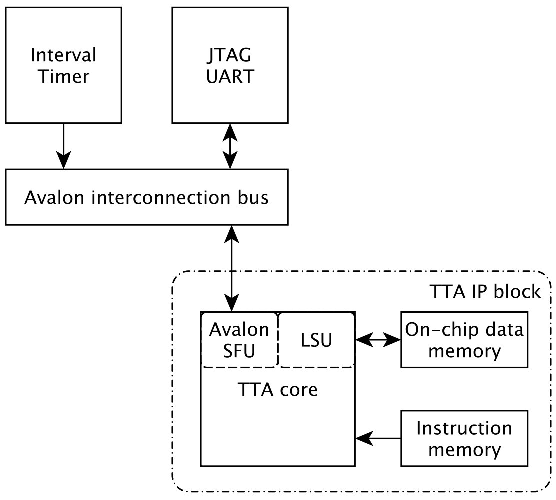


Figure 6.5: A SoPC design with a TTA which accesses the Avalon components using the Avalon SFU. In this case, both the instruction and data memories are inside the TTA IP block and hidden from the rest of the system.

local memory access. This is the same result as with the stand-alone CRC TTA as shown in Table 6.2.

6.5 Koski Integrator

The Koski Integrator was evaluated in [36]. The setup, which was used in Koski Integrator test, is presented in Fig. 6.6. The Nios II/f is the master processor in the system and the CRC TTA IP block is utilized as a CRC accelerator. Nios II/f executes the eCos [47] real time operating system and utilizes HIBI for Inter-process Communication (IPC).

The CRC calculation is implemented as an IPC function call through HIBI. The IPC realization on the Nios II was implemented as a device driver called *crc_tta_drv* which implements the communication protocol between the Nios II and the TTA. Respectively, the main program of the CRC TTA was also modified to implement the counterpart to this communication protocol. A sequence diagram of the communication is illustrated in Fig. 6.7. The Nios II first initializes the driver by setting a callback function which will be called when the CRC result is ready. This allows the Nios II to perform other task meanwhile the TTA calculates the CRC checksum. The CRC evaluation is started by calling the calculate function of the driver which notifies the TTA by sending the size of the data to be evaluated. The TTA then initializes a HIBI channel for receiving the data and acknowledges the Nios II. Then the Nios II will initialize a DMA transfer for sending the data and execution returns to its main program. When the CRC checksum is ready, it is sent to the Nios II. Receiving the CRC checksum interrupts the Nios II execution and gives the con-

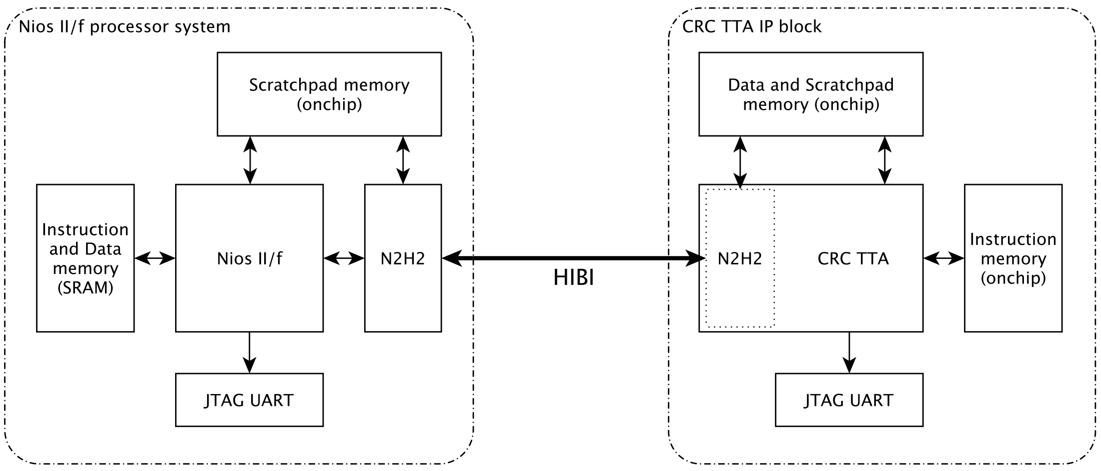

Nios II/f processor system
Scratchpad memory (onchip)
Instruction and Data memory (SRAM)
Nios II/f
N2H2
JTAG UART
HIBI
CRC TTA IP block
Data and Scratchpad memory (onchip)
N2H2
CRC TTA
Instruction memory (onchip)
JTAG UART

Figure 6.6: The design used for testing Koski Integrator [36].

trol to the *crc_ tta_ drv* which uses the given callback function to deliver the CRC checksum to the main program.

The CRC TTA processor was wrapped to a Koski compatible IP block using the Koski Integrator which requires the architecture to include the HIBI LSU stored in Koski specific HDB. The CRC TTA IP block was imported to the Kactus design tool which was used to assemble the design depicted in Fig. 6.6. The Nios II/f processor in the system was a ready made component in the Koski IP library and its clock

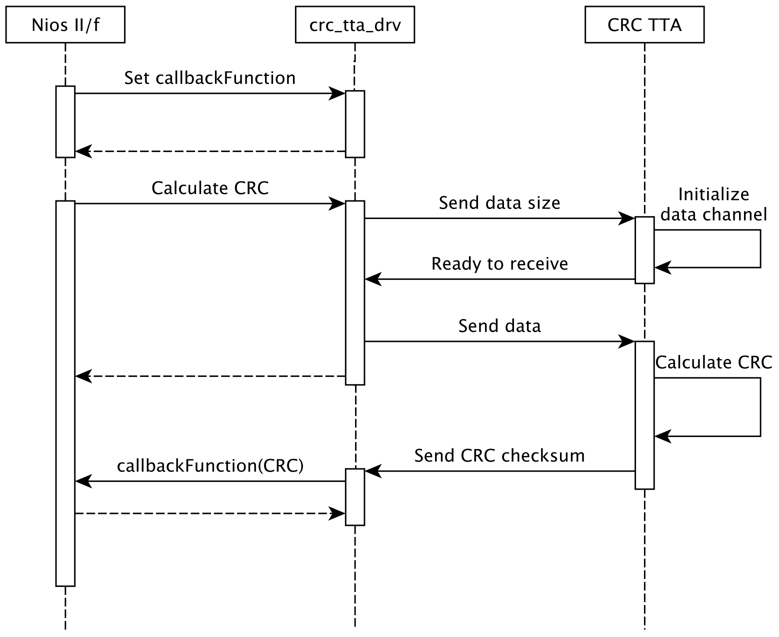


Figure 6.7: Sequence diagram of the communication between the Nios II/f and the CRC TTA for calculating a CRC checksum

Table 6.3: Synthesis results of the Koski test system for Stratix II. The target clock frequency was set to 50 MHz

Entity	**LE usage**
Whole system	13 908
TTA IP block	3 296
Nios II/f system	5 616

Table 6.4: Test case execution times from different verification levels measured with stand-alone Stratix II TTA.

Verification level	**Execution time / s**	**Execution frequency / MHz**
Desktop PC	0.002	2660.00
Arch. simulation (ttasim)	1.500	1.40
RTL simulation (Modelsim)	175.000	0.01
FPGA programming (Stratix II)	16.000	-
FPGA execution (Stratix II)	0.021	100.00

frequency was set to 50 MHz. This was chosen as the clock frequency of the whole system. The Koski tools were used to create a top level VHDL implementation of the system which was then synthesized for the Stratix II. The synthesis results are listed in Table 6.3. The LE usage increase in TTA is mostly due to the HIBI LSU which includes the N2H2 DMA controller. The ready made Nios II/f configuration used in this test case includes a variety of peripheral controllers which increase the resource usage compared to the Nios II/f used in Section 6.3. Finally, the system was executed on the FPGA to verify its correct behavior.

6.6 Summary

This section showed that the created Platform Integrators are able to successfully integrate TTAs into a stand-alone processor setup and to a wrapped IP block. The results gathered in Table 6.2 show the scalability of the TTA as an ASIP template. Tailoring the processor architecture resulted in 24.5 times better performance compared to the minimalistic TTA while the resource usage only increased by a factor of 1.45. The performance comparison against the Nios II/f soft-core demonstrates the capability of the TTA to exploit instruction level parallelism better than the scalar RISC based processor.

The verification flow presented in Chapter 5 was utilized in testing the Platform Integrators. Table 6.4 presents the verification execution times of the stand-alone

CRC TTA test case from the different verification levels. These results motivate the desire to find bugs and faults at the highest possible level since the execution time increases when moving to lower levels. For example, the execution time of the RTL simulation is over two orders of magnitude slower than the architecture simulation which makes a significant difference. The FPGA execution breaks the trend of the slowdown, but then again, the debugging capabilities on FPGA are limited. It is also worth noticing that the FPGA execution time was only 10 times slower than the execution time on the desktop PC while there is 26.6x difference in the execution clock frequency.

Verification speed is important, especially, when more complex designs and programs are verified. For example, it could take a few minutes to execute and verify a multicore video encoder ASIP on an FPGA, but executing the same test in the architecture simulator might take several hours and the RTL simulation could take days to finish. This emphasizes the importance of using the right verification level for debugging problems.

7. CONCLUSIONS

This thesis introduced a platform integrator framework for TCE ASIP design toolset. The purpose of this framework is to automate and speed up the integration process of TTA processors to a target FPGA. Another use case for the framework is to create TTA IP components which can be utilized in System-on-Chip design tools. The requirements and design of the Platform Integrator were described and documented in the thesis. The Platform Integrator design aims to be FPGA vendor independent to allow support for various FPGA vendors to be added in future.

Three Platform Integrator components were implemented and introduced in this thesis. The first one, Stratix II Integrator, realizes the stand-alone FPGA integration and the two other, the Avalon Integrator and the Koski Integrator, implement the TTA IP wrapper integration. The hardware units for supporting these Platform Integrators were also implemented for the thesis.

The second part of this thesis documented the verification flow of TTA processors. The top-down approach of the multi-level verification flow makes it possible to discover faults at the highest possible design abstraction level which helps in isolating the source of the fault. A new verification tool, TTA Unit Tester, was created to improve the verification process. This tool is utilized to test and verify individual function units and register files before generating the processor implementation. Thus, the tool adds another verification level to the flow. The thesis presented the requirements and design of TTA Unit Tester tool.

The implemented Platform Integrators were verified to work using a test application. These test cases demonstrated that the Platform Integrators create synthesizable and functional hardware. In order to increase the performance, a tailored TTA processor with custom operations was designed to match the test application. The test cases followed the described verification flow starting from the desktop PC execution and finally finishing to the FPGA execution. The verification execution times were measured from the different verification levels in order to emphasize the importance of finding the potential faults as early as possible in the verification flow. For example, the difference in the execution speed between the architecture simulation and the RTL simulation was over two orders of magnitude.

The goals of the thesis were reached but the platform integration framework still has room for improvements. For this thesis, only the support for Altera FPGA platforms was implemented. Although the framework was designed to be vendor independent, it has not been put to a proper test yet in this regard. Another point for improvement is the on-chip memory generation which relies on the Quartus II command line tools on the Altera specific platforms. This might cause problems for the code maintainability as the implementation depends on 3rd party tools. Therefore, an alternative solution would be desirable. As a future improvement, the memory generator components could also create memory hierarchies in addition to direct memory connections.

REFERENCES

[1] TCE: TTA-Based Codesign Environment. [Online] `http://tce.cs.tut.fi`.

[2] S.D. Brown. An overview of technology, architecture and CAD tools for programmable logic devices. In *Proc. IEEE Custom Integrated Circ. Conf.*, pages 69–76, May 1994.

[3] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2:135–253, Feb. 2008.

[4] Altera Corporation. *Stratix II Device Handbook, Volumes 1-2*, Jul. 2009. [Online] `http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf`.

[5] K. Keutzer, S. Malik, and A.R. Newton. From ASIC to ASIP: The next design discontinuity. In *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers and Processors*, pages 84–90, 2002.

[6] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Trans. Computer-Aided Design Integrated Circ. Syst.*, 26(2):203–215, Feb. 2007.

[7] J. Tong, I. Anderson, and M. Khalid. Soft-core processors for embedded systems. In *Proc. Int. Conf. Microelectronics*, Dhahran, Saudi Arabia, Dec. 16–19 2006.

[8] P. Yiannacouras, J. Rose, and J.G. Steffan. The microarchitecture of FPGA-based soft processors. In *Proc. Int. Conf. Compilers, Arch. and Synth. Embedded Syst.*, pages 202–212, 2005.

[9] Altera Corporation. *Nios II Processor Reference Handbook*, Dec. 2010. [Online] `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`.

[10] P. Yiannacouras, J.G. Steffan, and J. Rose. Application-specific customization of soft processor microarchitecture. In *Proc. Int. Symp. Field Prog. Gate Arrays*, pages 201–210, 2006.

[11] M. Labrecque, P. Yiannacouras, and J.G. Steffan. Scaling soft processor systems. In *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, pages 195–205, Apr. 2008.

[12] M. Keating and P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[13] S.J.E. Wilton and R. Saleh. Programmable logic IP cores in SoC design: opportunities and challenges. In *Proc. IEEE Conf. Custom Integrated Circuits*, pages 63–66, 2001.

[14] G. Martin and H. Chang. System-on-chip design. In *Proc. Int. Conf. ASIC*, pages 12–17, Oct. 2001.

[15] D. Flynn. AMBA: Enabling reusable on-chip designs. *Micro, IEEE*, 17(4):20–27, Jul. 1997.

[16] W. Badawy and G.A. Jullien, editors. *System-on-Chip for Real-Time Application*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[17] W. Wolf. *Modern VLSI Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2002.

[18] Altera Corporation. *SOPC Builder User Guide*, Dec. 2010. [Online] `http://www.altera.com/literature/ug/ug_sopc_builder.pdf`.

[19] Altera Corporation. *Avalon Interface Specification*, Aug. 2010. [Online] `http://www.altera.com/literature/manual/mnl_avalon_spec.pdf`.

[20] Altera Corporation. *Quartus II Handbook Version 10.1 Volumes 1-3*, Dec. 2010. [Online] `http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf`.

[21] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Trans. Embedded Computing Syst.*, 5:281–320, 2006.

[22] T. Koskinen. Metadata-based Automated Configuration of System-on-Chip. Master's thesis, Tampere University of Technology, Finland, Jun. 2009.

[23] Accelera Organization Inc., IP-XACT Technical Committee. [Online] `http://www.accellera.org/activities/ip-xact/`.

[24] The SPIRIT Consortium. *IP-XACT User Guide v1.2*, Jul. 2006.

[25] E. Salminen. *On Design and Comparison of On-Chip Networks*. PhD thesis, Tampere University of Technology, Finland, 2010.

[26] A. Kulmala. Multiprocessor system with general-purpose interconnection architecture on FPGA. Master's thesis, Tampere University of Technology, Finland, Aug. 2005.

[27] A.P. Chandrakasan and R.W. Brodersen. Minimizing power consumption in digital CMOS circuits. *Proc. IEEE*, 83(4):498–523, Apr. 1995.

[28] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[29] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[30] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala. Impact of software bypassing on instruction level parallelism and register file traffic. In *Proc. Int. Workshop Embedded Computer Syst.: Architectures, Modeling and Simulation*, pages 23–32, 2008.

[31] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proc. Int. Symp. Microarchitecture*, pages 303–312, 1995.

[32] O. Esko, P. Jääskeläinen, P. Huerta, C.S. de La Lama, J. Takala, and J.I. Martinez. Customized exposed datapath soft-core design flow with compiler support. In *Proc. Int. Conf. Field Programmable Logic and Applications*, pages 217–222, 2010.

[33] J. Mäntyneva. Automated Design Space Exploration of Transport Triggered Architectures. Master's thesis, Tampere University of Technology, Finland, Jul. 2009. `http://tce.cs.tut.fi/`.

[34] L. Laasonen. Program Image Generator and Processor Generator for Transport Triggered Architectures. Master's thesis, Tampere University of Technology, Finland, Apr. 2007. `http://tce.cs.tut.fi/`.

[35] Altera Corporation: DSP Development Kit, Stratix II Professional Edition. [Online] `http://www.altera.com/products/devkits/altera/kit-dsp-2S180.html`.

[36] O. Esko. Utilizing Transport Triggered Processors on FPGA-based system-on-chip. Bachelor's Thesis, Tampere University of Technology, Finland, Mar. 2011. `http://tce.cs.tut.fi/`.

[37] C. Pixley, N.R. Strader, W.C. Bruce, Jaehong Park, M. Kaufmann, K. Shultz, M. Burns, J. Kumar, Jun Yuan, and J. Nguyen. Commercial design verification: methodology and tools. In *Proc. Int. Test Conf.*, pages 839–848, Oct. 1996.

[38] V. Korhonen. Tools for Fast Design of Application-specific Processors. Master's thesis, Tampere University of Technology, Finland, Jan. 2009. `http://tce.cs.tut.fi/`.

[39] P. Jääskeläinen. Instruction Set Simulator for Transport Triggered Architectures. Master's thesis, Tampere University of Technology, Finland, Sep. 2005. `http://tce.cs.tut.fi/`.

[40] C. Pixley, N.R. Strader, W.C. Bruce, J. Park, M. Kaufmann, K. Shultz, M. Burns, J. Kumar, J. Yuan, and J. Nguyen. Commercial design verification: methodology and tools. In *Proc. Int. Test Conf.*, pages 839–848, Oct. 1996.

[41] GHDL: Open Source VHDL simulator. [Online] `http://ghdl.free.fr/`.

[42] Mentor Graphics ModelSim: Advanced Simulation and Debugging. [Online] `http://model.com/`.

[43] Oracle: VirtualBox home page. [Online] `http://www.virtualbox.org`.

[44] IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks–Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Section One. *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pages c1–597, 2008.

[45] M. Barr: Example implementation for calculating CRC. [Online] `http://www.netrino.com/code/crc.zip`.

[46] Altera Corporation. *Embedded Peripherals IP User Guide*, Dec. 2010. [Online] `www.altera.com/literature/ug/ug_embedded_ip.pdf`.

[47] eCos: eCos home page. [Online] `http://ecos.sourceware.org/`.