TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAKARI LAHTI

INTEGRATION OF PROCESSOR AND SYSTEM-ON-CHIP TOOLS

Master's thesis

Examiners:
Professor Timo D. Hämäläinen,
Dr. Tech. Erno Salminen.
Examiners and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 9 October 2013.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LAHTI, SAKARI: Prosessori- ja system-on-chip-työkalujen yhteiskäyttö
Diplomityö, 60 sivua, 0 liitesivua
Kesäkuu 2014
Pääaine: Ohjelmoitavat alustat ja laitteet
Tarkastajat: Prof. Timo D. Hämäläinen, TkT Erno Salminen
Avainsanat: TTA, TCE, IP-XACT, Kactus2, C-to-VHDL, FPGA, SoC

Siirtoliipaistuun arkkitehtuuriin (engl. transport-triggered architecture, TTA) perustuvat suorittimet tarjoavat tehokkaan välimallin ratkaisun luotaessa IP-komponentteja System-on-chip -piireihin. TTA-suorittimien avulla suunnittelutyö on huomattavasti vaivattomampaa ASIC-lähestymistapaan verrattuna, ja toisaalta taloudellisempi ja tehokkaampi toteutus on mahdollinen kuin käyttäen yleiskäyttöisiä suorittimia.

Tässä diplomityössä tutkitaan tapoja nopeuttaa suunnitteluvuota käytettäessä TTA-suorittimia SoC-suunnittelutyössä. Esitetyt vuot yhdistävät “TTA-based Co-design Environment” -työkalupaketin (TCE) käytön Kactus2 IP-XACT-suunnitteluympäristön kanssa. IP-XACT-standardi ja Kactus2-työkalu helpottavat eri valmistajien tarjoamien IP-komponenttien toisiinsa yhdistämistä ja konfiguroimista, kun taas TCE-työkalut tarjoavat nopean ja tehokkaan reitin C-kielestä VHDL:ään.

Tässä työssä esitellään kolme TTA-käyttötapausta: valmiiksi tehtynä kiinteänä kiihdyttimenä, yleiskäyttöisenä suorittimena, ja räätälöitynä sovelluskohdistettuna suorittimena. Lisäksi työssä käydään läpi instanssikohtaisen datan käsittelyä IP-XACT:ssa. Suunnitteluvuot käydään askel askeleelta läpi jokaisen käyttötapauksen osalta, esimerkkitapaus esitellään, ja jokaiseen askeleeseen käytetty suunnitteluaika arvioidaan.

Vuot sisältävät 15-18 askelta ja niiden yhteydessä käytetään 8-12:ta eri ohjelmaa käsitellyistä ohjelmistotyökalupaketeista. Jos C-lähdekoodi ja IP-XACT-kirjasto ovat valmiina, insinööri voi toteuttaa FPGA-pohjaisen monisuoritinlaitteen alle 4 tunnissa ilman mittavaa aiempaa laitteistosuunnittelukokemusta. Tulosten perusteella esitellään lisäkehitysehdotuksia TCE-työkaluihin ja Kactus2:een.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
LAHTI, SAKARI: Integration of Processor and System-on-Chip Tools
Master of Science Thesis, 60 pages, 0 Appendix pages
June 2014
Major Subject: Programmable platforms and devices
Examiners: Prof. Timo D. Hämäläinen, Dr. Tech. Erno Salminen
Keywords: TTA, TCE, IP-XACT, Kactus2, C-to-VHDL, FPGA, SoC

Transport-triggered architecture (TTA) processors provide an efficient middle-ground in creating intellectual property (IP) components for system-on-chip (SoC) designs. Using TTAs, the design effort is greatly reduced compared to ASIC approach, and a more economic and efficient implementation is possible than when using a general purpose processor.

This Thesis examines ways to accelerate the design flow when using TTA processors in SoC designs. The proposed flows combine the use of the TTA-based Co-design Environment (TCE) tool set and Kactus2 IP-XACT design environment. The IP-XACT standard and the Kactus2 tool make it easy to integrate and configure IP components from multiple vendors, whereas the TCE tools provide a fast and efficient path from C to VHDL.

The Thesis presents three use cases for TTA: as a ready-made fixed accelerator, a general purpose processor, and a tailored application-specific processor. Moreover, management of instance-specific data in IP-XACT is discussed. For each use case, the design flows are presented in detail step-by-step, a case example is presented, and the design time spent on each step is evaluated.

The flows contain between 15 and 18 steps and use between 8 and 12 different program tools from the studied tool sets. Provided that C source codes and IP-XACT library are available, a non-HW oriented engineer can implement an FPGA based multiprocessor product in less than 4 hours. Based on the results, further development suggestions for the TCE tools and Kactus2 are made.

PREFACE

This Master of Science Thesis was written in the Department of Pervasive Computing at Tampere University of Technology during the fall of 2013 and the spring of 2014. The bulk of the research took place during the spring of 2012. The research results were published in the following conference article and this Thesis expands and elaborates upon the paper:

Lauri Matilainen, Sakari Lahti, Otto Esko, Erno Salminen, Timo D. Hämäläinen. Integration of TTA processor tools to Kactus2 IP-XACT design flow. Norchip, Copenhagen, Denmark, November 12-13 2012. 6 pages.

I wish to thank Lauri Matilainen, Joni-Matti Määttä and Antti Kamppi for valuable help with Kactus2 and Otto Esko and Pekka Jääskeläinen for similar help with the TCE tools. I am also thankful to my parents and friends for being there and enriching my life during all these years. Most of all, I am grateful to my supervisor Timo D. Hämäläinen who was exceptionally patient, friendly and helpful during the struggle to give birth to this Thesis.

I see great potential in both Kactus2 and the TCE tools. Hopefully, this Thesis provides at least some modest benefit to their developers.

May 14, 2014

Sakari Lahti

CONTENTS

1 Introduction....1
2 Overview of TCE and Kactus2....5
2.1 TCE Overview....5
2.1.1 Principles of a TTA processor....5
2.1.2 TCE tools....7
2.2 Kactus2 overview....9
2.2.1 The IP-XACT/IEEE1685 Standard....11
2.2.2 Kactus2 IP-XACT extensions....14
2.3 About instance management in IP-XACT....17
2.3.1 Creating a new IP-XACT component from a template or draft....17
2.3.2 Completing instance-specific data into IP-XACT design....18
2.4 Summary of the Tools....18
3 TTA AS a fixed accelerator IP block....20
3.1 IP block design....22
3.2 SoC design....25
4 TTA as a general purpose processor....26
4.1 Processor design....29
4.2 SoC design....31
5 TTA as an ASIP....33
5.1 Template processor design....36
5.2 ASIP and SoC design....37
6 An illustrative example with crc....39
6.1 Template component creation....39
6.2 Using the component in a SoC design....41
7 Design flow Evaluation....46
7.1 TTA as a fixed accelerator IP block....46
7.2 TTA as a general purpose processor....48
7.3 TTA as an ASIP....50
8 Conclusions....53
8.1 General remarks....53
8.2 Suggestions for future development of the tools....54
8.2.1 Suggested modifications to the TCE tools....55
8.2.2 Suggested modifications to Kactus2 55
References....57

LIST OF TERMS AND ABBREVIATIONS

ASIP	Application-specific instruction-set processor. A processor whose instruction set is tailored to benefit a certain application.
CPU	Central processing unit. A hardware component that processes the instructions of a computer program performing basic arithmetical, logical and input/output operations.
Custom FU	Custom function unit. An FU that has been customized to perform a specific task that is not needed in all TTAs.
DSP	Digital signal processing. Mathematical manipulation of information in discrete form used to modify or improve it in some way.
FPGA	Field-programmable gate array. An integrated circuit designed to be configured by a designer after manufacturing.
FU	Function unit. A basic component within a TTA processor performing operations when it is triggered by incoming data.
GPP	General purpose processor. A processor designed for running arbitrary applications.
GPU	Graphics processing unit. An electronic circuit designed to rapidly manipulate memory to accelerate the creation of images intended for output to a display.
HDL	Hardware description language. A computer language used to program the structure, design, and operation of electronic circuits. Examples: VHDL and Verilog.
HLL	High-level language. A programming language with strong abstraction from the details of the HW platform.
HLS	High-level synthesis. An automated design process that interprets an algorithm and creates digital HW that implements it.
HW	Hardware. Any physical digital electronic component.
IP block/component	Intellectual property block/component. A reusable HW component that is the intellectual property of one party.
IP-XACT	An XML format that defines and describes electronic components and their designs.

IP-XACT library	An organized collection of IP-XACT metadata of HW and SW components and designs saved on disk that can be accessed from Kactus2.
Kactus2 library	See IP-XACT library.
Metadata	“Data about data.” Contains concise information about the important aspects of data.
MP-SoC	Multiprocessor SoC. A SoC design that contains multiple processors.
OSAL	Operation set abstraction layer. A database in TCE that contains the static properties and simulation behavior of operations.
RISC	Reduced instruction set computing. A CPU design philosophy with relatively few and simple instructions.
RTL	Register-transfer level. A design abstraction which models a synchronous digital circuit in terms of the flow of digital signals between HW registers.
SoC	System-on-chip. An integrated circuit that integrates all components of an electronic system into a single chip.
SW	Software. Any program or part of program that can be processed by a CPU.
TCE	TTA-based co-design environment. A tool set for designing and programming customized TTA processors.
TTA	Transport-triggered architecture. A CPU design in which programs directly control the internal transport buses of processors.
VLIW	Very long instruction word. A processor architecture designed to take advantage of instruction level parallelism.
VLNV	Vendor-library-name-version. A system for identifying HW and SW components by their vendor, library, name and version number.
VLSI	Very-large-scale integration. A process of creating an integrated circuit by combining thousands of transistors into a single chip.
XML	Extensible markup language. A markup language that defines a set of rules for encoding documents in a format that is both human- and machine-readable.

LIST OF FIGURES

Figure 1: A SoC architecture for MPEG-4 encoder with performance profiling support [2, p. 2]....2
Figure 2: Comparison of choices for implementing an application function....2
Figure 3: A sample TTA processor architecture [21, p. 11]....6
Figure 4: The TCE design flow [21, p. 9]....7
Figure 5: The general design flow and scope of Kactus2....10
Figure 6: Kactus2 views of the library, SoC design and system design. The SoC design contains processors, buses and other components from the library. The processors have SW components mapped to them in the system design where also interprocessor SW communication channels are defined....11
Figure 7: IP-XACT-based design flow....12
Figure 8: IP-XACT components, design and design configurations....13
Figure 9: Component and design hierarchy in IP-XACT....14
Figure 10: Top-level aspects in Kactus2 design flow....14
Figure 11: a) Standard IP-XACT way for including SW. b) Consequence: HW component library grows....15
Figure 12: a) SW as IP-XACT components and mapping from HW to SW. b) Consequence: composed designs from generic HW and SW components....15
Figure 13: Mapping between HW and SW with system design....16
Figure 14: Completing a template component....17
Figure 15: The template processor architecture as seen in ProDe. Input ports with “X” denote ports that trigger the FU....39
Figure 16: The values given in the ports editor in Kactus2. Only the mandatory values have been entered for these simple ports....40
Figure 17: The values given in the bus interface editor for the clock bus in Kactus2....40
Figure 18: The component view of the finished TTA template component....41
Figure 19: The library (in the left) and the HW design with two TTA instances....42
Figure 20: The Kactus2 views editor for the simple TTA core....43
Figure 21: The system design mapping SW to TTAs. The blue boxes represent the TTAs and the green boxes the SW....43
Figure 22: The architecture of the TTA customized for CRC calculation....44

LIST OF TABLES

Table 1: The various TCE tools used in this work. The tools either have a graphical user interface (GUI) or execute from the shell command line (CLI). Output file format is also specified. [21, p. 10].....8
Table 2: Standard IP-XACT objects and Kactus2 extended IP-XACT objects. Objects that are not relevant to this Thesis are marked with an asterisk.....16
Table 3: Summary of the TCE tools and Kactus2.....19
Table 4: The design flow phases of a fixed accelerator IP block design and integration. Phases F1-F11 contain the IP block design and phases F12-F15 integration into a SoC design and implementation on a FPGA.....20
Table 5: The design flow phases of a general purpose TTA processor design and integration. Phases G1-G9 contain the processor design and phases G10-G18 the SoC design and implementation on a FPGA.....27
Table 6: The design flow phases of a template TTA processor creation.....34
Table 7: The design flow phases of a custom processor creation and integration.....34
Table 8: Time-usage estimations of each phase in the design flow of TTA as a fixed accelerator IP block (as in Table 4).....46
Table 9: Time-usage estimations of each phase in the design flow of TTA as a general purpose processor (as in Table 5).....48
Table 10: Time-usage estimations of each phase in the design flow of a template TTA processor creation (as in Table 6).....50
Table 11: Time-usage estimations of each phase in the design flow of a custom TTA processor creation and integration (as in Table 7).....51
Table 12: Summary of the properties of the design flows.....53

1 INTRODUCTION

System-on-chip (SoC) integrates dozens of intellectual property (IP) components into a single chip, typical applications including telecommunication and multimedia [1]. Designing complex SoCs requires an efficient reuse of existing hardware (HW) and software (SW) components and modern design tools which enable automation. For example, the interface between HW and SW, verification, design space exploration, and product data management are very important tasks. At the same time, the required processing requirements are high while the available power budget is very limited, especially in mobile devices.

In addition to fixed HW accelerators which handle the most demanding processing, there are from a few to dozens of programmable processors, and consequently a large fraction of design costs are associated with embedded SW. Such a heterogeneous processing approach provides an affordable balance between performance and development costs.

Figure 1 presents an example of a typical SoC architecture [2]. The SoC implements an MPEG-4 video encoder application on FPGA with multiple processors (CPUs) that run the application SW, two HW accelerators (ME and DCT-Q-IDCT) for resource-intensive parts of the encoding algorithm, a memory controller, a resource manager (RM) to arbitrate the processors' access to the HW accelerators, and a hardware monitor to collect data on the performance of the accelerators. In this SoC, the various components are connected by an on-chip communication network called HIBI [3] which is implemented by the wrappers connected to each component. Furthermore, the HW accelerators require their own wrappers since they have interfaces that are incompatible with HIBI. The wrappers also manage the dataflow considerations to and from the accelerators. Even this relatively simple SoC demonstrates how the design process can grow complex and thus error-prone unless rigorous designing principles are followed with efficient auxiliary program tools.

Today's SoCs are typically manufactured with 28 nm technology and can contain one or more central processing units (CPU), a graphics processing unit (GPU), internal memory blocks, external memory controllers, digital signal processors (DSP), and various external interfaces for different industry standards. In addition, application domain specific modules and accelerators are common. To give an example of a modern SoC is Qualcomm's Snapdragon 800 MSM8974 targeted for smartphones and tablets, which contains a 2.3 GHz quad-core CPU, a 450 MHz GPU, a 600 MHz DSP, a 32-bit dual-channel LPDDR3 memory controller, and radio modules for WiFi, Bluetooth and GPS

among other components [4]. This SoC has seen wide use in the mobile devices industry.

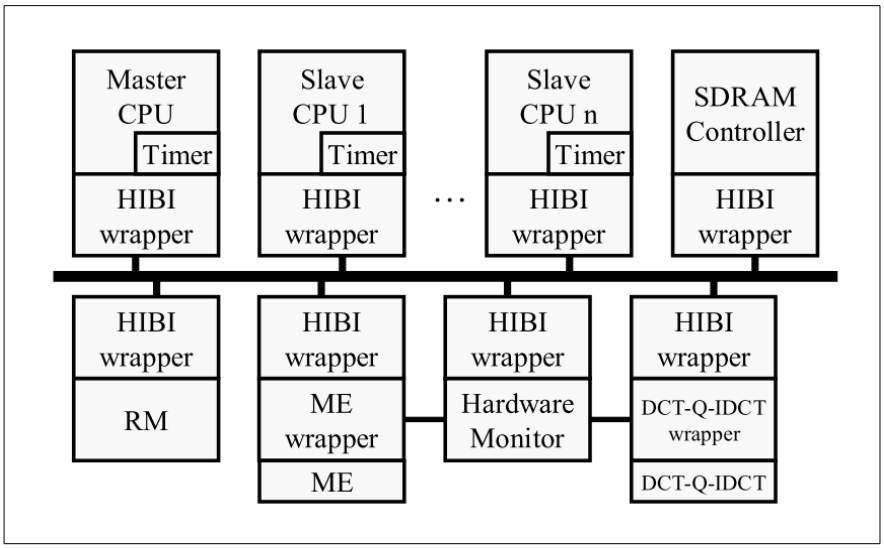

Figure 1: A SoC architecture for MPEG-4 encoder with performance profiling support [2, p. 2].

Field-programmable gate array (FPGA) is a common SoC platform whose customizability makes it an attractive option for many design problems. Figure 2 shows the different approaches to implement an application on an FPGA. The starting point is a description of the application as an executable model in C language that abstracts the communication between application functions. The first, quickest implementation option is the compilation of the C code into NIOS [5] or other general purpose processor (GPP). If the performance is not satisfactory, the next option is to use an application specific instruction-set processor (ASIP), such as a transport-triggered architecture (TTA) processor [6; 7]. The last and most laborious option is to implement bare HW by converting the algorithms into synthesizable register-transfer level (RTL) description with VHDL.

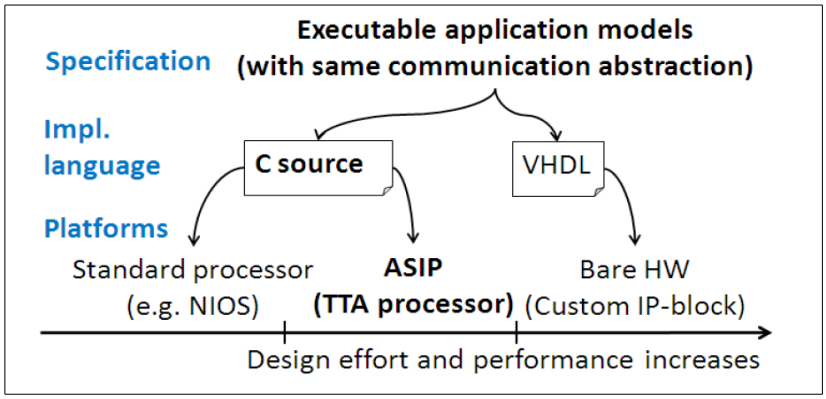

Figure 2: Comparison of choices for implementing an application function.

As a rough estimate, if the GPP should take 1 day, ASIP could take 2 days and RTL more than 10 days to implement. Based on a set of experiments, if the performance of NIOS is normalized to 1, a TTA can be 4x faster and RTL 2−50x faster [2]. Thus, the ASIP approach is often the most practical option since it provides a good effort-performance ratio and completely avoids or greatly reduces VHDL coding.

There are plenty of existing SoC design flows, which can be divided into high-level synthesis (HLS) approaches and integration environments. In HLS, a designer describes the application in a high-level language (HLL) such as C, SystemC or MATLAB, and the HLS tool maps this description to HW constructs as efficiently as possible. In addition to faster time-to-market, another advantage of these design methodologies is that they usually require less HW design expertise, allowing people with SW engineering skills to produce HW IP components with adequate performance. These flows are sometimes referred to as ”C-to-VHDL”. A popular way of implementing HLS for FPGAs is to use soft-core processors with varying degrees of configurability [7; 8; 9; 10; 11]. For example, about 4x speedup over basic reduced instruction set computing (RISC) processor was reported in [10].

Numerous SoC integration environments exist and recently they have started utilizing the IEEE1685/IP-XACT XML metadata standard as well [12; 13; 14; 15]. The goal is to packetize all reusable IPs into a library to simplify integrating and configuring them. The necessary information includes, among others, port interfaces, file sets, available parameters, and optional generator scripts. Tools can then automatically generate a structural top-level VHDL, compilation scripts, header files for SW developers, and so on. Furthermore, the vendor/library/name/version (VLNV) identifiers of all IP-XACT objects aid in version and product data management. The most important objects are component (for example CPU), interface definitions (for example AMBA [16]) and design which is a hierarchical description of the component instances and their connections.

This Thesis combines the two approaches (HLS and SoC integration environments) by focusing on enhancing the SW integration in Kactus2 design flow [17; 18; 19] using TTA processors. The approach is based on the IP-XACT standard which is originally purposed for HW IP block integration. Kactus2 expands on this, allowing capturing for example the structure of the SW stack, its mapping into processors and memory map design. The goal is to help SW engineers to implement FPGA applications by integrating TTA tools into the Kactus2 IP-XACT design environment.

The TTA tool package developed at the Tampere University of Technology is called the TTA-based Co-design Environment (TCE). Previously, there has been no formal research on how to combine the TCE and Kactus2 design flows. This work considers three different use cases of designing and integrating TTA processors with the TCE tools and Kactus2. The first case is where the TTA processor is purposed as a fixed accelerator IP block. That is, the processor will perform a single task in the most efficient

way and it is not re-programmable. In the second case, the TTA is designed to be a GPP similar to NIOS. In the final, and most important case, the TTA is stored in IP library as a template component that can be reused and customized as an ASIP. The aim of the Thesis is to provide clear and efficient design flows for integrating the TCE and Kactus2 use in these three cases.

The rest of the work is organized as follows. Chapter 2 provides an overview of the TCE tools and Kactus2. Chapters 3, 4 and 5 present the use cases and the related design flows of TTA as a fixed accelerator IP block, GPP and ASIP, respectively. Chapter 6 contains an illustrative example of the ASIP design flow. Chapter 7 evaluates the design flows from the perspective of time-usage, and finally, Chapter 8 provides the conclusions drawn from the work.

2 OVERVIEW OF TCE AND KACTUS2

This work combines the use of two separately developed, freely available, open-source design environments: TTA-based Co-design Environment and Kactus2 used for IP integration.

2.1 TCE Overview

2.1.1 Principles of a TTA processor

The TCE tools are based on TTA processors [6], which follow the principle that computation occurs as a side effect of data transports (it can be thought that TTA is an extreme version of a RISC processor having only one instruction, *move*). Each function unit (FU) within the processor has a specific input port that triggers computation within the unit when data arrives in the port. The FUs are connected by one or more transport buses, and each instruction word of a program defines the data transfers within each bus. This allows for strong instruction level parallelism like in the very long instruction word (VLIW) processors [20]. The number of simultaneous operations within the processor is limited only by the number of transport buses. Furthermore, unlike in conventional processors, it is not always necessary to write output back to a register file as the output of a FU can be directly sent to the input port(s) of the FU that needs it. This is called software bypassing, which can significantly relieve register file port pressure and consequently energy consumption.

Thanks to transport triggering, the control logic of the processor is usually simpler than in conventional processors. Many of the control decisions that are normally made at run time can be fixed when compiling the program. This, however, means that a program compiled for a certain TTA processor is unlikely to work on any other processor since the compiled program code assumes a fixed FU and bus architecture.

Figure 3 shows the basic architecture of a TTA processor. The sockets in the Figure define which FUs are connected to which transport buses, and furthermore they enable and disable the connections within, according to the program that is being run. Essentially, a TTA program is just a set of instructions about which connections are enabled on each clock cycle. Data flows based on the currently enabled connections and operations are triggered within FUs when new data arrives in their triggering input ports.

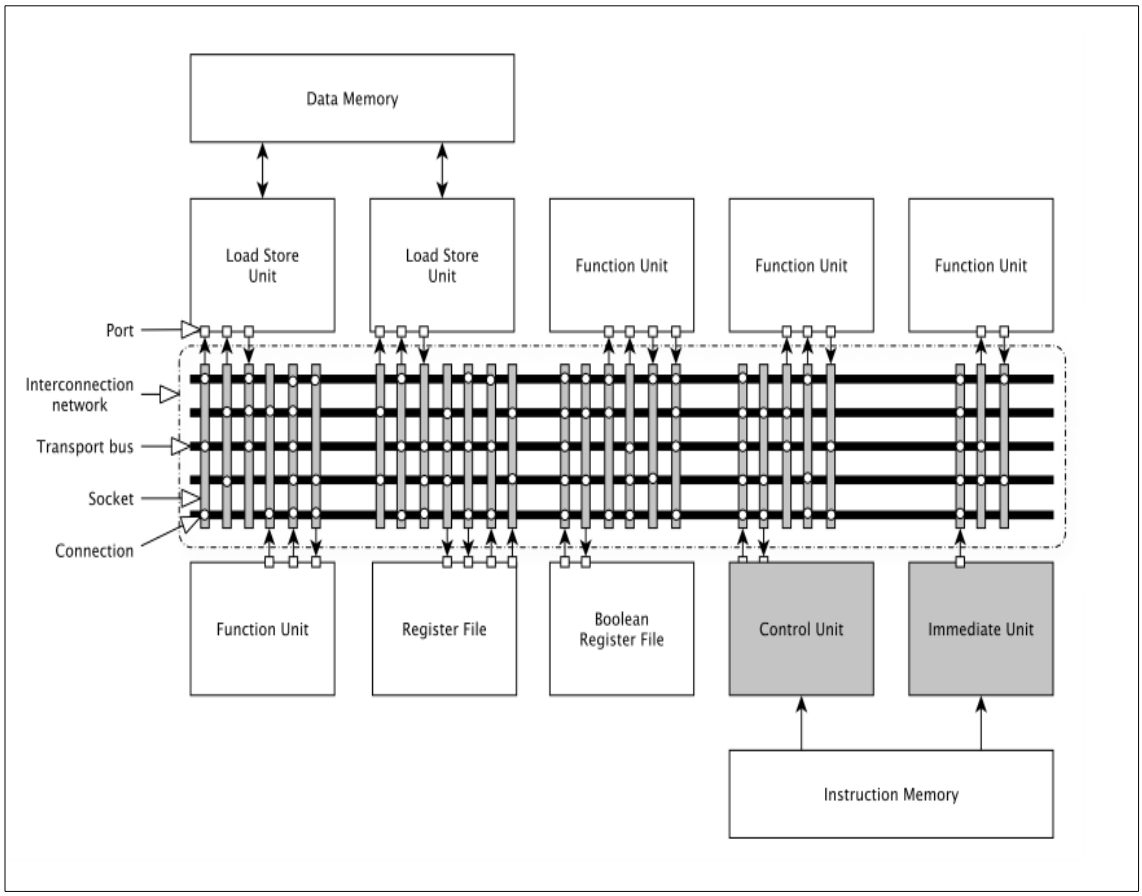

Figure 3: A sample TTA processor architecture [21, p. 11].

The TTA paradigm is ideal for ASIPs. The processor design model is flexible and straightforward as the main tasks are selecting the FUs and their connections, which are easy to modify thanks to the TCE tools that are discussed in the next subsection. Custom HW accelerators can be created to handle computation-intensive operations, but this is not necessary with all applications.

A comparison of performance between a simple TTA processor and various soft-core processors for several applications was performed in [10]. The TTA processor that was implemented with the TCE tools and the soft-core processors (NIOS II/f and two MicroBlaze configurations) were synthesized and run on FPGA. The results showed that the TTA processor outperformed the soft-core processors provided by major vendors on average by a factor of 1.5 to 2 when measuring speedup but even 4x speedups were measured.

TTA processors are best suited for signal processing and data flow -type applications that can be run to completion without external interrupts. Applications requiring interrupts are difficult to support since moves of one operation can span multiple instruction words, so all of the processor state cannot be represented by the registers alone, but also includes state inside execution units. Fortunately, the drawbacks of TTAs are usually not issues with ASIPs where there is often no need to run multitasking operating systems.

2.1.2 TCE tools

The TCE tools are being developed at the Department of Pervasive Computing at Tampere University of Technology. Their goal is to enable an effortless design flow of programmable TTAs and to provide compilers for certain HLLs (currently C/C++ and OpenCL) to avoid writing arduous assembly code for TTA processors.

Figure 4 shows the TCE design flow which is described in an abbreviated form here. More comprehensive information can be found in [21].

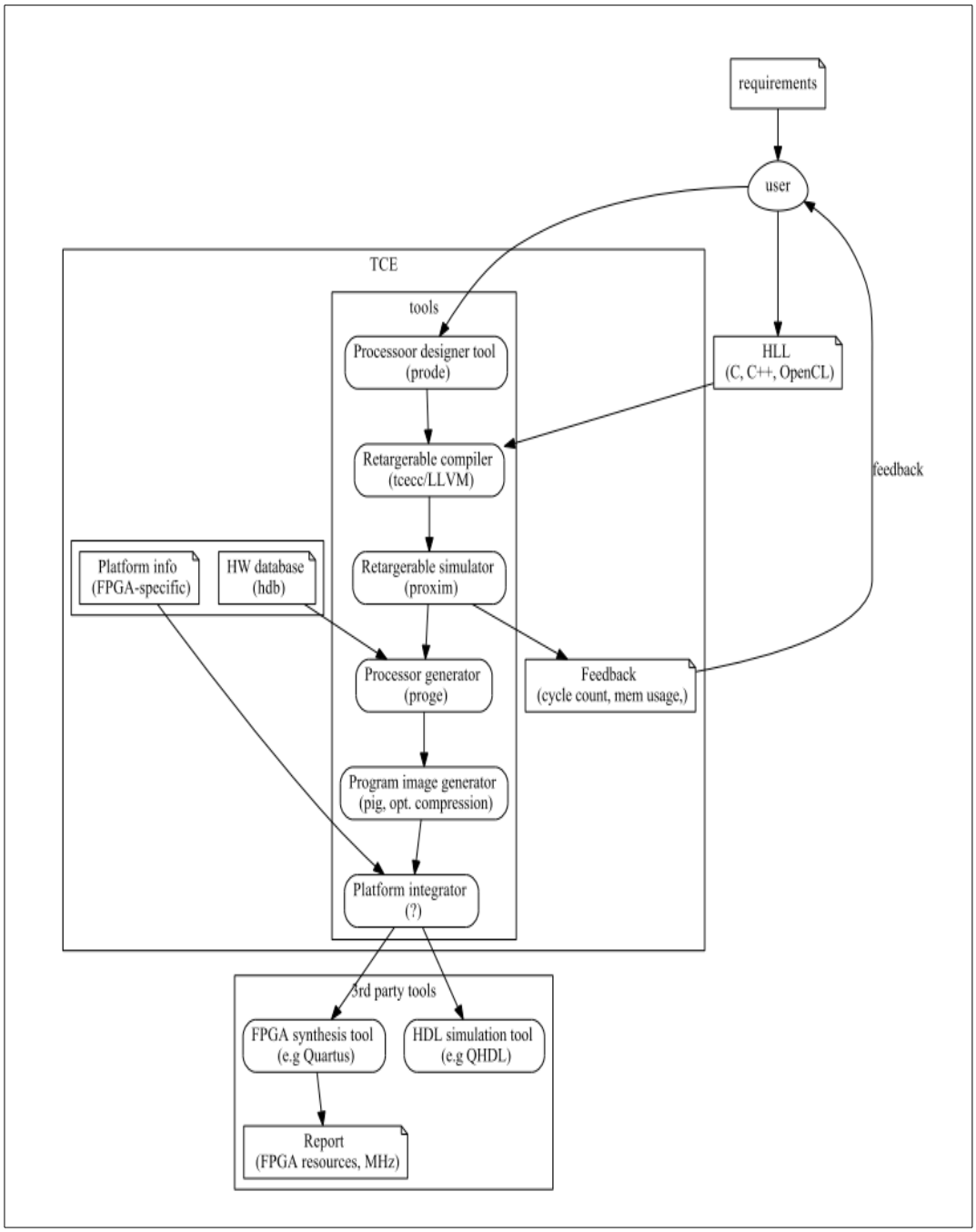

Figure 4: The TCE design flow [21, p. 9].

In the first phase, the user creates a starting point processor architecture using the *Processor Designer*[1] tool (*ProDe*), or selects an existing architecture. He then compiles the given HLL program for the processor using a retargetable[2] compiler (*TCECC*). The user can then use a processor simulator program (*Proxim* or *TTASim*) to find data about cycle count and FU utilization among other things. Based on this feedback, he can change the architecture until he finds one with sufficient performance. The *Automatic Design Space Explorer* tool (*Explore*) can be used to partially automate this iterative process. After this, he can select the VHDL implementations for the FUs from the HW database (HDB) using *ProDe*.

The same FU can have different implementations in the HDB which decouples the functionality and implementation of the FUs. This is a useful property since the database can have different implementations for different FPGA families, among others.

The *Processor Generator* (*ProGe*) creates synthesizable VHDL code based on the architecture and implementation mapping, and it also includes the *Platform Integrator* which can be used to create synthesis settings and integrate the TTA with the memory components of a given FPGA board. The *Platform Integrator* can also create an IP-XACT description of the TTA with the *KoskiIntegrator* option.

In the last phase, the actual program binaries are generated using the *Program Image Generator* (*PIG*). It can create plain binary as well as various FPGA vendor specific RAM formats. Synthesis and simulation are done using FPGA specific third-party tools.

Other TCE tools include the *Operation Set Abstraction Layer Editor* (*OSEd*) which can be used to add information on various custom operations that the FUs may perform to the corresponding database. The *HDBEditor* is used to modify hardware databases that contain the VHDL files of FU implementations. *Estimate* can be used to approximate physical properties of TTAs, such as area, maximum clock frequency and energy consumption. Table 1 summarizes the TCE tools.

Table 1: The various TCE tools used in this work. The tools either have a graphical user interface (GUI) or execute from the shell command line (CLI). Output file format is also specified. [21, p. 10]

Tool	**Purpose**	**Type**	**Output file(s)**
ProDe	Define FUs, registers, interconnects of TTA core	GUI	ADF
TCECC	Compile software	CLI	TPEF
Proxim	Simulate TTA cores	GUI	report

1 Throughout this work, the various TCE tools are identified with italic font.

2 Retargetability means that the compiler needs to know the target architecture which can change from compilation to compilation.

Tool	Purpose	Type	Output file(s)
TTASim	Simulate TTA cores	CLI	report
OSEd	Operation set abstraction layer database editor	GUI	operation database
Estimate	Estimate physical properties of TTAs	CLI	report
HDBEditor	HW implementation database editor	GUI	HDB
ProGe / generateprocessor	Generate HDL	CLI (+GUI)	VHDL
PIG / generatebits	Generate program image	CLI	for example MIF
Platform Integrator	Interface with memories, generate IP-XACT	CLI	XML, project files

2.2 Kactus2 overview

Kactus2 is a tool set for designing embedded products, especially FPGA-based multi-processor SoCs (MP-SoC) [22]. It uses the IEEE1685/IP-XACT XML metadata and design methodology, but extends the IP-XACT usage to upper product hierarchies and HW/SW abstraction with Multicore Association MCAPI [23].

Kactus2 enables drafting and specifying from scratch block diagram blueprints for product boards, chips, SoCs and IPs and get them stored in IP-XACT format. It also allows packetizing IP for reuse and exchange by, for example creating “electronic data sheets” of existing IPs for library as templates and blocks ready for integration. Finally, it can be used in designing MP-SoC products by creating HW designs with unlimited hierarchy, and system designs that map SW to HW. Product creation is further supported by tools that automatically generate everything needed for HDL synthesis and SW build, such as top-level VHDL and Altera Quartus II project files. However, IP functionality and binaries/executables cannot be created directly in Kactus2, although there is a clear path from Kactus2 to other tools.

Figure 5 depicts the scope of Kactus2 and the key tasks within it. The starting point can be formal, executable models or other means of documentation on required functionality. System level design creates HW and SW partitions using for example model based tools. HW and SW components are acquired, either by creating new or reusing existing ones from a library. Metadata can be used as specification for new components.

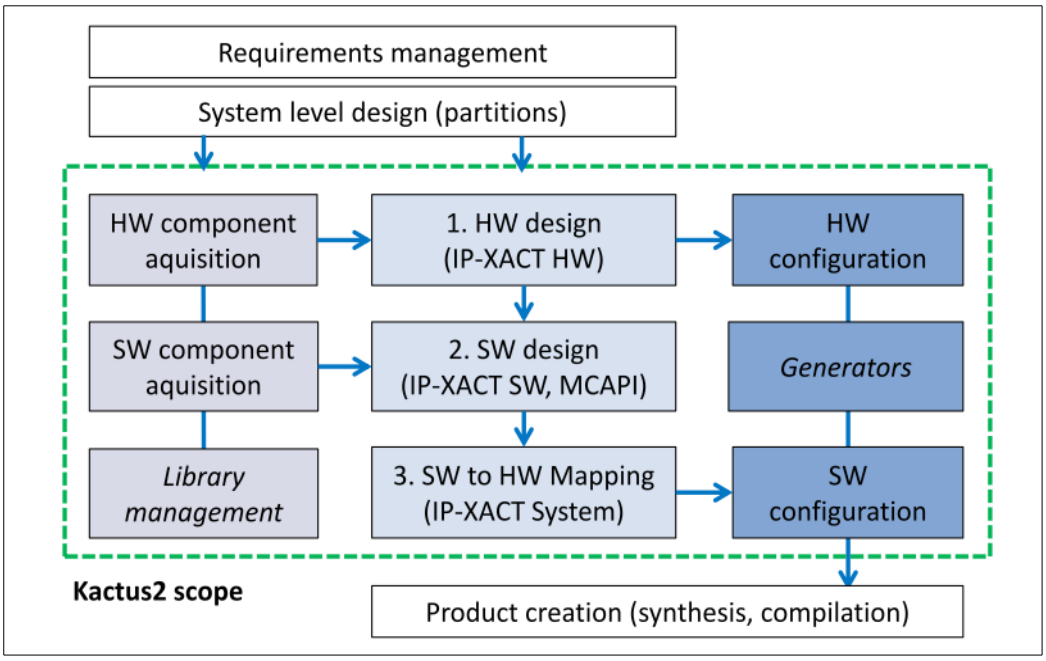

Figure 5: The general design flow and scope of Kactus2.

The integration order is following: First HW component integration, then SW architecture design, mapping of SW components to HW instances, and finally configuring both HW and SW components for product creation. The flow describes primarily SoC design, but the phases can be applied to other levels of product hierarchy as well. For example, when "board" is the outcome, design information is related to PCB schematic, layout, part lists, silkscreens, and process-related information like test setups, test programs and test data patterns.

Figure 6 shows sample views from Kactus2 related to a SoC design. In the left is shown the IP-XACT library that contains all HW and SW components and designs. In the middle is the SoC (HW) design containing processors and other components along with their connecting buses. In the right is the system design where the processors have SW components mapped on them and SW communication channels are defined. Furthermore, each HW and SW component in the library or in a design can be opened and configured separately.

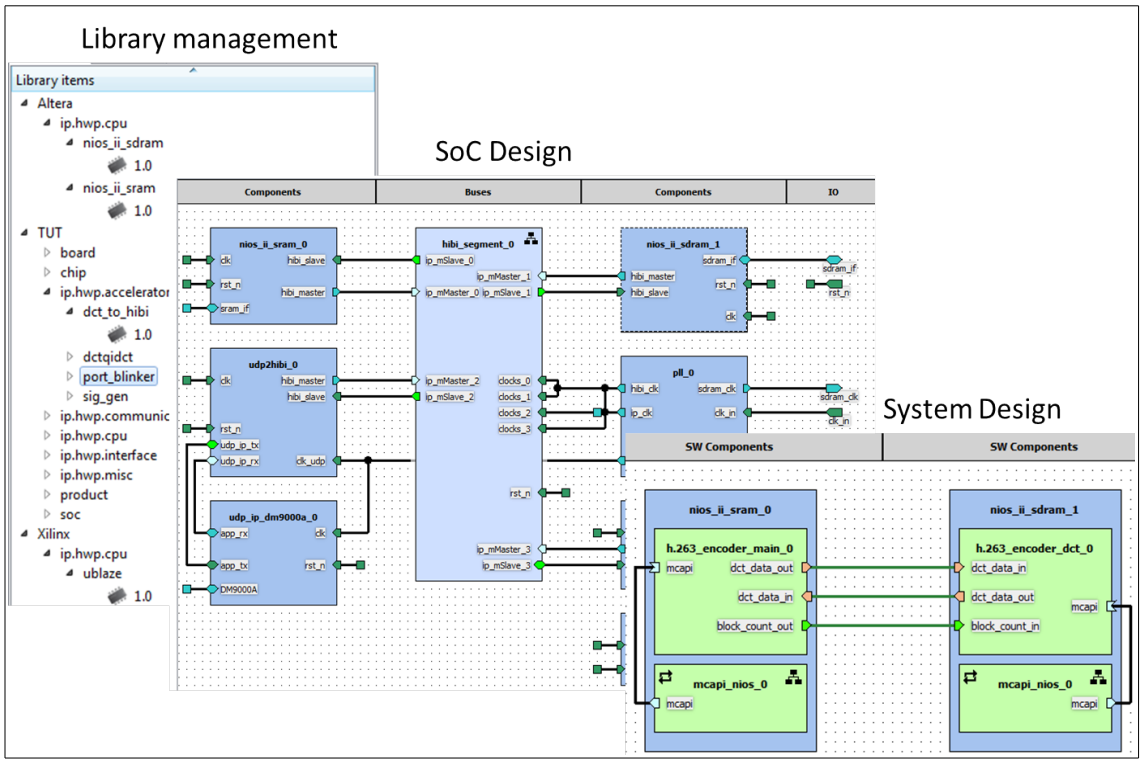

Figure 6: Kactus2 views of the library, SoC design and system design. The SoC design contains processors, buses and other components from the library. The processors have SW components mapped to them in the system design where also interprocessor SW communication channels are defined.

2.2.1 The IP-XACT/IEEE1685 Standard

Kactus2 utilizes the IP-XACT/IEEE1685 standard that was first developed by the SPIRIT Consortium and approved as IEEE1685 in 2009. IP-XACT is an XML format that describes very-large-scale integration (VLSI) HW IP blocks to enable automated configuration and integration through tools. The scope of IP-XACT is on the IP and SoC levels, but in Kactus2 the standard is applied also to other levels, aiming at product level information management.

A core concept in IP-XACT is metadata which means data about data content. Kactus2 uses metadata for describing components and designs. Metadata for a component is a formal, vendor- and technology-independent description of the component that includes references to source files and other related information. Components are in practice HW blocks and SW code in different abstraction and granularity levels. For a design, metadata is a formal structural description that includes references to component metadata, tools, configurations and other design-related information.

Figure 7 summarizes the main IP-XACT design steps. The sources are encapsulated and separated from the IP description. This means that the HDL source code is embedded via links to the source file in metadata file. IP blocks are assembled together in a de-

sign, which is a structural description of the system. IP blocks as well as the design itself may have generic parameters, which are configured using generators that are typically scripts. The final configured IP-XACT design can be seen as “instructions” on how to create an executable. In Kactus2, IP-XACT metadata is also used backwards as specification for a new IP block not yet existing.

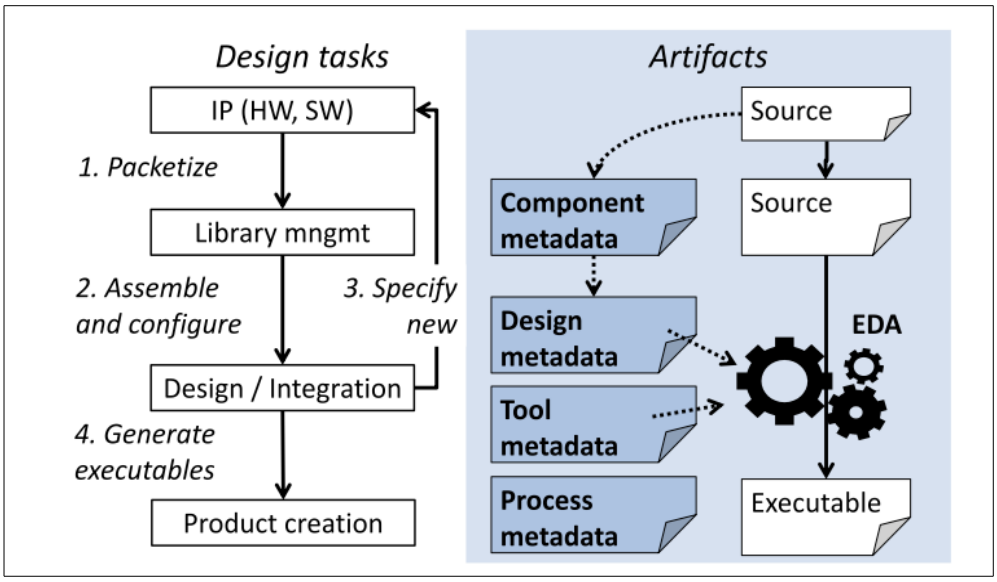

Figure 7: IP-XACT-based design flow.

IP-XACT objects are XML metadata files representing SoC components, structure, and configurations. IP-XACT design environment handles these objects, but not directly the source files. The final design including all components and their connections is also an object itself. IP-XACT objects are uniquely identified and referred to by a tuple {V,L,N,V} stating Vendor-Library-Name-Version. All VLNV tuples are unique, independent of what kind of IP-XACT objects they identify. The XML file name and its location on disk are not defined themselves.

A SoC design flow may use several models for the same IP block or design, starting from high-level abstract models down to implementation accurate models. Typically, models are separately stored and each might have a different description language (for example UML, SystemC, VHDL). IP-XACT can include all different abstraction level descriptions in one metadata object. Instead of several separate objects, there can be only one with several options. This helps keeping the library coherent and help automating the path from specification to implementation. IP-XACT also supports mixed abstraction levels at the same time.

An *IP-XACT component* is a general placeholder describing all IP block types like processors, memories, accelerators and building blocks for buses and various interfaces. A component contains independent elements that can be referenced between each other.

Views are used to represent different roles of the component. Example views include "RTL implementation", "documentation", "simulation" and "SW implementation."

Components are connected using *bus interfaces*, *bus definitions*, and *abstraction definitions*. Bus interface defines a grouping of ports, with ports allowed to be included in multiple different bus interfaces. Bus definition specifies general bus properties like whether bus is addressable and what kind of connections are allowed. Abstraction definition defines logical bus signals and constraints related to them such as bus width and direction of logical signals. A *port map* defines the mapping between physical and logical signals.

File sets and *file set groups* are folder and file collections that can be associated to views. Example file set groups include "application," "interrupt," and "device driver." File sets include information about used tools, description languages and instructions on how to handle files.

An *IP-XACT design* is like a traditional schematic of components. It describes a list of component instances, their configuration, and connections to each other. For a design, several *design configurations* exist for different purposes. Figure 8 illustrates the relationship between components, designs and design configurations.

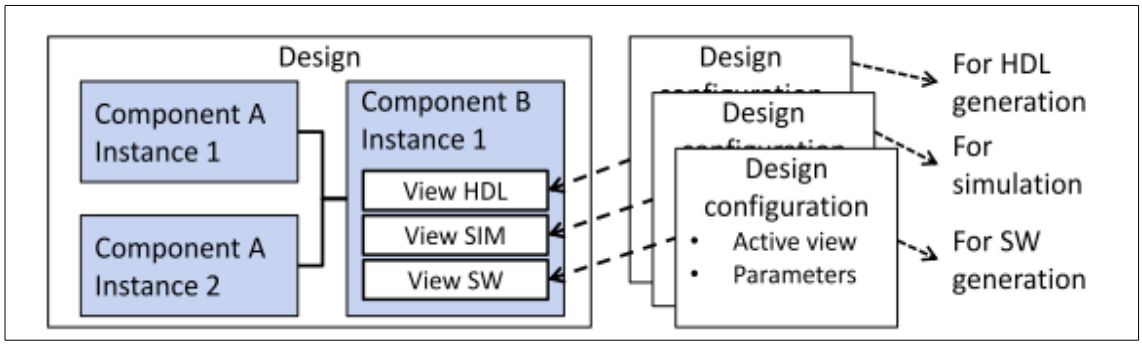

Figure 8: IP-XACT components, design and design configurations.

A *hierarchy of designs* is implemented as follows (see also Figure 9):

- IP-XACT designs never refer to other designs. Instead, a design always refers to components that are instantiated into design.
- The design must be wrapped inside a component in order to use it as a sub-design. Thus, an IP-XACT component refers to an IP-XACT design.
- An IP-XACT design can always be used as an IP-XACT component.

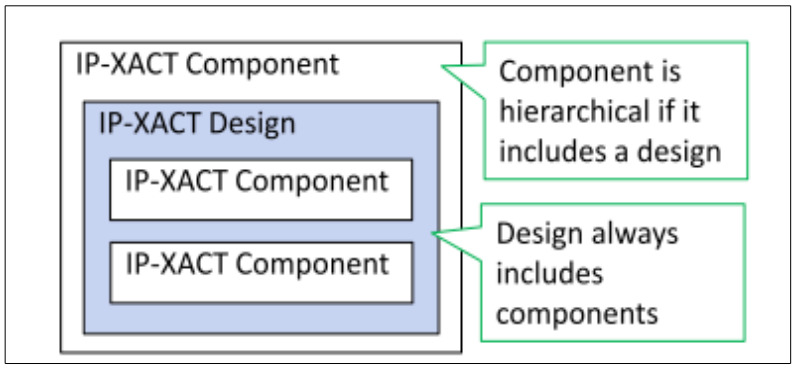

Figure 9: Component and design hierarchy in IP-XACT

2.2.2 Kactus2 IP-XACT extensions

IP-XACT was originally purposed for HW descriptions. Kactus2 extends this to use SW and system-level descriptions to apply IP-XACT to its design flow and to make IP-XACT easier to use. Information on these extensions can be found for example in [24] and [25], but the key points are summarized below.

The Kactus2 design flow manages three top-level aspects:

- *Product hierarchy:* This defines the scope of the work.
- *Implementation:* All objects are categorized according to HW, SW, system (HW and SW mappings), and communication abstraction.
- *Firmness:* Each library object is one of the following: Mutables are reusable components that can be modified. Templates must be saved to a new version prior to use. Fixed have all parameters determined for frozen product releases.

Standard IP-XACT elements are used to describe the above top-level aspects. Comparison of the original scope of IP-XACT and these aspects is shown in Figure 10.

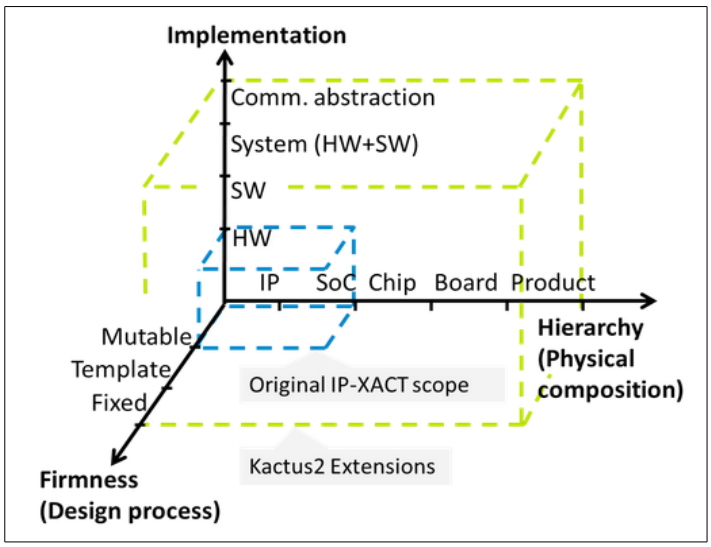

Figure 10: Top-level aspects in Kactus2 design flow.

A standard IP-XACT HW component may include references to SW through *model views* and file sets pointing to files on disk. In the worst case, a new HW component version is needed every time the component is instantiated in a new design with new related SW (Figure 11). In Kactus2, this is avoided by creating IP-XACT objects also for *SW components*.

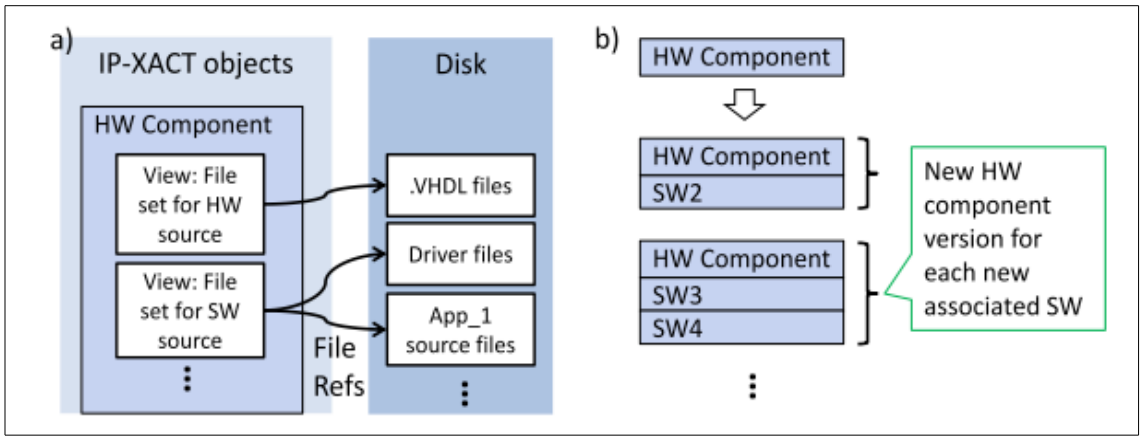

Figure 11: a) Standard IP-XACT way for including SW. b) Consequence: HW component library grows.

With separated HW and SW components, designs can be composed without adding design-specific files and elements to HW components as shown in Figure 12. However, referencing to SW from HW components is still allowed, but now *object references* are used (references between VLNVs). Using this mechanism, the mapping from HW component to SW component is made through *SW design*. SW design may include only one instantiated SW component, or there can be several SW components of its own hierarchy.

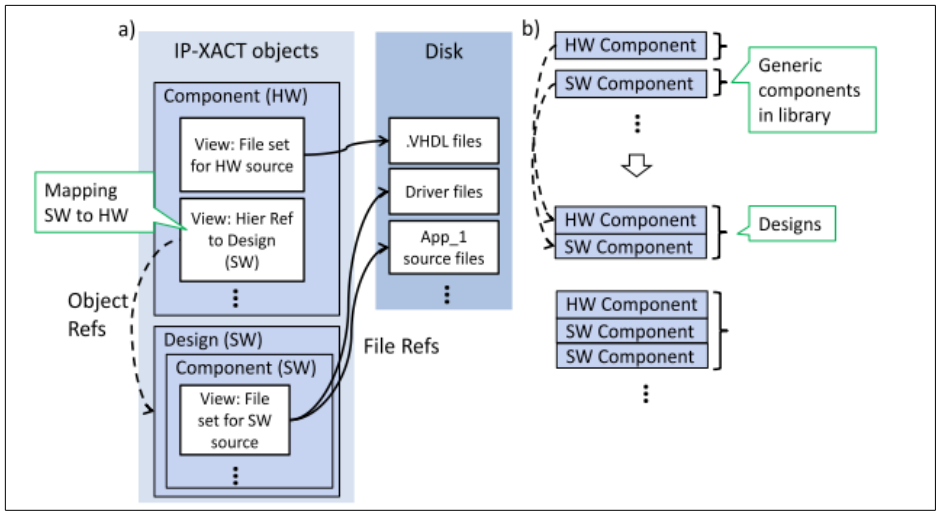

Figure 12: a) SW as IP-XACT components and mapping from HW to SW. b) Consequence: composed designs from generic HW and SW components.

To instantiate HW and SW components on the same IP-XACT design, a *system design* is defined. It is a SW design that includes mappings of SW component instances to HW component instances. Since IP-XACT does not specify HW/SW mappings, a *model parameter* is added to each SW component for the mapping that is configured in system design (Figure 13).

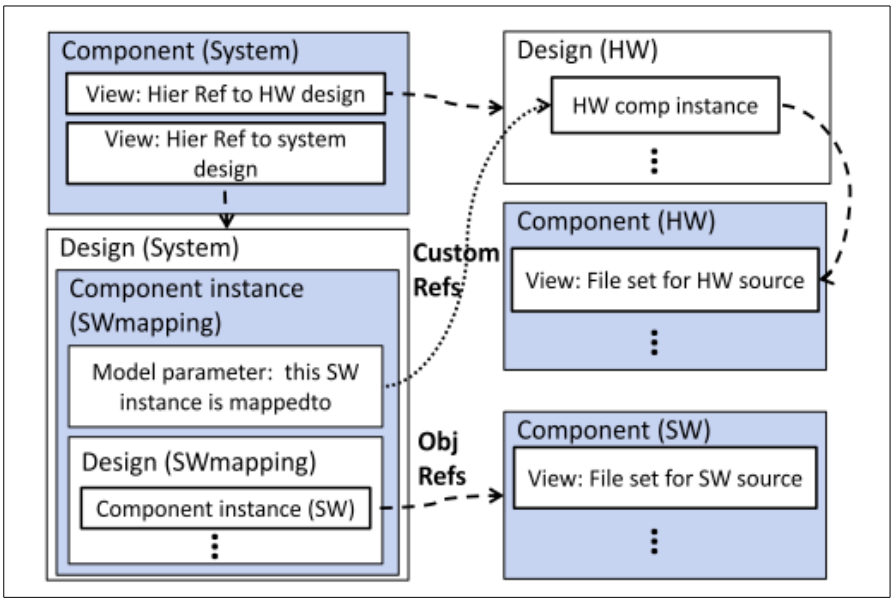

Figure 13: Mapping between HW and SW with system design.

To summarize, the mapping of SW to HW can be done either by component hierarchical model view specified in a HW component (“from HW to SW”) or by instantiated component model parameter specified in a system design (“from SW to HW”).

Table 2 collects the standard IP-XACT objects and Kactus2 extended IP-XACT objects together. Objects that are not relevant to this work have been omitted from the explanations in this chapter.

Table 2: Standard IP-XACT objects and Kactus2 extended IP-XACT objects. Objects that are not relevant to this Thesis are marked with an asterisk.

IP-XACT standard objects	**New Kactus2 objects**
HW component, HW design	SW component, SW design
HW design configuration	API definition* (SW)
Bus definition	COM definition* (SW, HW)
Abstraction definition	System design (SW architecture mapped to HW)
Generator chain*	System component (SW architecture)

2.3 About instance management in IP-XACT

The standard IP-XACT integration flow starts from reusable components that are instantiated, connected, and configured in an IP-XACT design. Unfortunately, the standard has problems with instance-specific definitions. Although generics and connections are well defined, for example address spaces and SW components are not. The basic question regards which values to store to the reusable component and which to the instances. Two approaches are considered in the following.

2.3.1 Creating a new IP-XACT component from a template or draft

Figure 14 depicts a design process with templates or incomplete components in the beginning. A template is a component that is never used as such in a design; it is like an abstract base class in programming. In this example, the IP-XACT library includes a few TTA template components that define only the information common to all instances that are generated from them, like generators, and which connection is used with an on-chip network, and if an external memory interface is used (SRAM or DRAM). User makes a copy of the template, gives it a new VLNV, and instantiates it in the HW design.

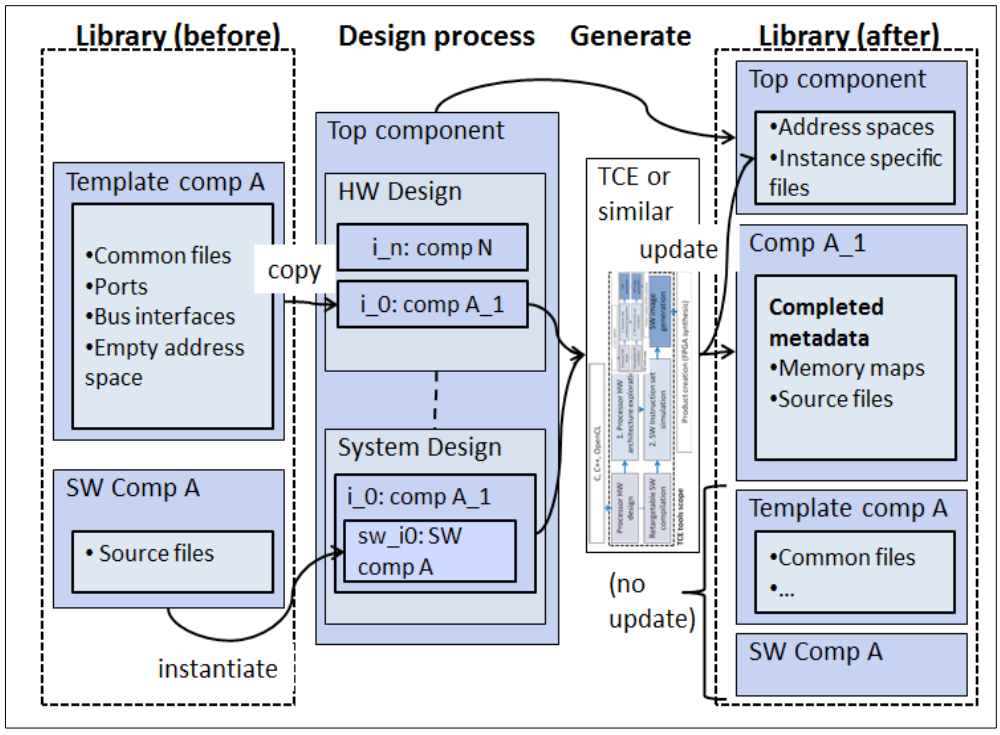

Figure 14: Completing a template component.

Kactus2 also offers an option to ”draft” new components. User just adds an empty component, connects it to others and defines a VLNV identifier. The necessary interface types are detected automatically and serve as part of the requirements for the IP development team. This is one way of creating a TTA template component for the library.

In Figure 14, source files are added to the new component A_1 later by a component specific generator. However, in the worst case the library will contain as many components as there are instances in various designs.

2.3.2 Completing instance-specific data into IP-XACT design

Figure 14 also depicts the second approach where instance-specific data is stored to the place of the instantiation, that is to the SoC. As a specific example, one can consider the address space definition that can be added to master interfaces in IP-XACT. It defines what kind of segments the component can access using a certain interface. Thereby, address space naturally depends on the design where the component is instantiated. Address space should not be confused with the memory map that describes which registers of a component are visible to others, which is usually not instance-specific information.

Again, the designer creates a HW design with all the necessary IPs followed by an optional system design phase. The example assumes that template comp A has an empty address space defined at first. The segments of the address space are then determined based on the other components in the HW design. The complete address space is used in SW compilation and stored to a top-level HW component (for example SoC) which refers to the HW and system designs. The same principle applies to SW mapping data.

Moreover, if VHDL files of an IP are automatically generated based on a few instance-specific parameters (for example data width, cache size, #FUs), they could be stored into the top component’s file set as well. The reusable component holds only the generator script, interfaces, parameter list, and documentation. Each instance defines the parameter values for the generator which produces the implementation files. This approach encapsulates the TTA-instance-specific values and files to the SoC component without augmenting the library with numerous components.

2.4 Summary of the Tools

Table 3 contains a summary of the primary properties of the TCE tools and Kactus2 for quick reference.

Table 3: Summary of the TCE tools and Kactus2.

	TCE tools	**Kactus2**
Purpose	Designing and programming customized processors based on the transport triggered architecture	Designing, specifying and managing embedded products, especially MP-SoCs.
Platform	Linux	Linux and Windows
Developer	Tampere University of Technology	Tampere University of Technology
Home page	http://tce.cs.tut.fi/index.html	http://funbase.cs.tut.fi/#main
License	MIT License	GPL2 General Public License
Implementation language	C, C++, Python	C, C++/Qt5
Lines of code	217,000	274,000

The lines of code includes comment lines. The tools are constantly evolving so the most current information can be found on their respective web sites.

3 TTA AS A FIXED ACCELERATOR IP BLOCK

Thanks to the TCE tools, one can efficiently develop various fixed TTAs each tailored to a certain application. These can be saved to the IP-XACT library, and the integrator can drag and drop them into a SoC design like any other IP component. There is only minimal if any processor configuration by the user. The processor's IP-XACT component metadata includes the VHDL source files for HW, and SW source code and memory image maps of the program it executes. This use case is natural in a situation where the processor design and SoC design are done by a different person, although both can be designed by the same engineer as well. The design flow for this use case is described after Table 4 which summarizes it.

Table 4: The design flow phases of a fixed accelerator IP block design and integration. Phases F1-F11 contain the IP block design and phases F12-F15 integration into a SoC design and implementation on a FPGA.

#	**Phase**	**Input**	**Tools**	**Output**
F1	SW application design and testing on a PC *or* reuse	Application specification documents *or* reused code	SW code writing tools, testing environment	Platform-independent SW source code
F2	Initial TTA processor design	HW specification documents, requirements of SW	*ProDe*	Initial processor ADF
F3	Program compilation for initial TTA	Initial processor ADF, source code files	*TCECC*	Initial TPEF
F4	Design space exploration	Initial ADF, Initial TPEF	*Proxim*, *TTASim*, *ProDe*, *TCECC*, *OSEd*, *Explore*	Final processor ADF

#	**Phase**	**Input**	**Tools**	**Output**
F5	C code modification and compilation for TTA	Platform-independent SW source code	SW code writing tools, *TCECC*	TTA-tailored C source code files, final TPEF
F6	*Optional:* VHDL implementation for FUs if not already in HDB	FU requirements	VHDL writing tools, *HDBEditor*	VHDL files for FUs, edited HDB
F7	FU implementation mapping	Final ADF, HDB	*ProDe*	IDF
F8	Estimation of physical properties	Final ADF, IDF, final TPEF	*Estimate*	Estimation results
	Either F9 if the estimation results are satisfactory or back to F4 if not			
F9	*Either F9.a or F9.b*			
F9.a	Processor, memory controller, and top-level entity generation with *Platform Integrator*	Final ADF, IDF, final TPEF, memory specifications	*Platform Integrator*	Top-level and hierarchical VHDL files of the processor and memory controllers, top-level entity VHDL, FPGA synthesis project files
F9.b.1	Manual processor generation	Final ADF, IDF	*ProGe*	Top-level and hierarchical VHDL files of the processor
F9.b.2	Manual memory controller generation	Memory specifications	VHDL writing tools or target platform's memory controller generator	Memory controllers' VHDL files
F9.b.3	Manual top-level entity VHDL generation	Top-level VHDL files of the processor and memory controllers	VHDL writing tools	Top-level entity VHDL
F10	Memory image generation	Final ADF, Final TPEF	*PIG*	Instruction and data memory images in chosen format
F11	*Either F11.a or F11.b*			
F11.a	IP-XACT component creation (if *KoskiIntegrator* was not used in F9)	All files created in the previous steps, documentation	Kactus2 component creation wizard	IP-XACT component of the TTA in library

#	Phase	Input	Tools	Output
F11.b	IP-XACT component augmentation (if *KoskiIntegrator* was used in F9)	IP-XACT file created by *KoskiIntegrator*, all files created in the previous steps, documentation	Kactus2	Augmented IP-XACT component of the TTA in library
F12	SoC design	IP-XACT files of the TTA and other components in the design	Kactus2	Kactus2 HW design
F13	Top-level VHDL generation of the HW design	Kactus2 HW design	Kactus2	Top-level VHDL of the HW design
F14	FPGA synthesis project file generation	Kactus2 HW design	Kactus2 (for Quartus II) or target platform's SoC designing program	FPGA synthesis project files
F15	FPGA synthesis and programming	All the VHDL files, memory image files, FPGA synthesis project files	FPGA synthesis and programming tools	The entire design synthesized and programmed on FPGA

3.1 IP block design

The initial part of the flow is the same as in Section 2.1.2, but it is described in more detail here. An even more detailed description can be found in [21], along with information on how to use the various tools mentioned here.

The flow begins with writing or reusing a C program code that performs the specified functionality of the accelerator block. The code doesn't need to be customized for TTA at this point, and indeed it should be tested on a workstation PC for correct functionality. If in the final product, the processor takes external input from for example a program running on another processor, it should be provided by the program itself at this point. In other words, the external I/O is emulated in the program. If possible, dummy values may be used.

When the SW code is ready, a starting point architecture of the TTA is created. This is done using the *ProDe* tool. The starting point architecture should contain at least the

minimum resources required to run the program. External I/O functionality can be omitted at this point. The end result is saved in an architecture definition file (ADF).

In the next phase, the program is compiled for the architecture using the *TCECC* compiler. The C source code files and the ADF file are given as parameters to the compiler, and the compiled TTA program is saved in the TTA program exchange format (TPEF) file. This is not a bit image that can be run on a physical platform, but it is used in simulating the program on the target TTA.

There are two instruction set simulator options, either *Proxim* that has a graphical user interface or *TTASim* which is command-line based. Both provide statistics about the cycle counts and utilization of FUs. The designer then modifies the TTA architecture based on the simulation results and specified performance and resource-usage requirements. The basic optimization options include modifying the algorithm itself at code level, adding more resources to the TTA (FUs, register files, buses), and adding custom operation FUs to the TTA. Details on performing these optimizations are outside the scope of this work but they can be found in [21, Section 3.1]. This design space exploration is an iterative process where different TTA architectures are created and tested until a satisfactory one is found. The *Explore* tool can be used to partially automate this process.

The designer should add any FUs related to external HW (for example various external I/O components) to the architecture at this point. If their implementation does not yet exist, he needs to add the related custom operations to the TCE's operation set abstraction layer (OSAL) using *OSEd*, map them to FUs and create the VHDL implementation of the FUs.

After finding a suitable architecture, the C code needs to be modified to be run on the TTA processor. Specifically, any input and output to/from the processor's internal custom FUs and I/O units needs to be handled in the C code using macros defined in the “tceops.h” library that includes a corresponding macro for each operation in OSAL. The modified code is compiled to create the final TPEF file.

The next phase is to create a VHDL implementation of the processor. First, each FU in the TTA needs to be mapped to a HW implementation. The implementations are stored in the TCE's HW database (HDB). There can be several different implementations for the same FU that differ in resource usage and target platform. If some FUs don't have an implementation in the HDB, they must be created by writing the VHDL and adding information on them to the HDB using the *HDBEditor* tool. The implementation mapping is performed using *ProDe*, and the result is saved in the implementation definition file (IDF).

With the implementation mapping, it is possible to use the *Estimate* tool to provide estimates on the energy consumption, die area, and maximum clock rate of the TTA. The final ADF, IDF and TPEF files are given as input parameters to the *Estimate* tool. If the output report contains unsatisfactory results, the flow returns to design space explo-

ration where the architecture, SW code and/or FU implementations are modified until the results improve, whereupon the designer can move on to the processor generation phase.

The TCE tools contain the *Platform Integrator* that is part of the *ProGe* tool. It supports automated integration of TTA cores to different FPGA platforms. The design flow divides in two parts here, depending on whether there is support for the target platform in the *Platform Integrator.*

If the *Platform Integrator* can be used, the final phases of the processor generation are rather straightforward. The user invokes the command-line version of the *ProGe* tool and gives to it as parameters the ADF, IDF and TPEF files, and parameters defining target platform and the type of instruction and data memory used. The *Platform Integrator* saves the processor's VHDL implementation files in a specified folder, creates memory controller VHDL, a top-level entity and even project files for FPGA synthesis with default pin-mappings. It should be noted that if the *Platform Integrator* is used, the processor's internal component implementations must be selected from the platform-specific options in the HDB. Currently, there is *Platform Integrator* support for Altera's Stratix II DSP Pro board and Stratix III FPGA Development Kit board. The *Platform Integrator* can also create an Altera SOPC Builder [26] component from a TTA processor, and a HIBI bus compatible processor.

If the *Platform Integrator* does not support the target platform, then the processor, the memory controllers, and the top-level VHDL entity must be created manually. The processor is generated either directly using *ProDe* or with the command-line *ProGe* tool. The ADF and IDF files are given as parameters and the VHDL implementation of the processor is saved in the target directory by the tool. The memory controllers, and the top-level entity connecting the TTA to the memory controllers and containing ports for external signals are created by hand. Another option is to use the *Platform Integrator* anyway, and then modify the automatically generated files to support the target platform.

The *PIG* tool is used to create a bit image of the program which can be uploaded to the target platform's memory for execution. It is invoked from the command-line with the “generatebits” command, and the ADF and TPEF file and the chosen instruction and data memory formats are given as parameters.

The *Platform Integrator* has an option that generates the IP-XACT file of the processor automatically. This *KoskiIntegrator* adds all the relevant data to the IP-XACT except for file sets containing source code and other documentation files which must be manually added with Kactus2. When *KoskiIntegrator*, which requires Altera FPGA and HIBI support, cannot be used, Kactus2 can create the IP-XACT file with an integrated component creation wizard. The user adds the relevant files (VHDL, program code, documentation, and so on) to the file sets and writes a description of the component in the corresponding field. The IP-XACT component is then automatically created based

on the top-level VHDL component. The created component is saved in the IP-XACT library for integration and use in designs. There, it is seen as any fixed IP block. All TTA instances created from the same component are identical and run the same code.

3.2 SoC design

The use of the previously created TTA IP block component is straightforward in Kactus2. The SoC designer simply drags and drops the component from the library to his HW design and connects it to other components. Its functionality is fixed so the only customization can be done on SoC level.

The top-level VHDL file of the HW design is created automatically by the Kactus2 VHDL generator. Kactus2 also creates the FPGA synthesis file automatically for Altera's Quartus II program. If it is not used, the project file must be created using the target platform's SoC designing program. The entire design is now ready to be synthesized and programmed on FPGA using the platform-specific tool.

4 TTA AS A GENERAL PURPOSE PROCESSOR

In this use case, the TTA processor is used like any GPP. Hence, in the library it is an IP-XACT component which includes a CPU element but not a fixed program image. The processor is instantiated in a HW design and SW code is mapped to it in a Kactus2 system design. Existing SW code can be reused from the library or it can be created from scratch. Once the SW code is mapped to the TTA instances, the *TCECC* and *PIG* tools are invoked for each processor-program pair to generate program images.

Each processor instance has identical HW, but may execute individual code. A designer may later modify and recompile the code. Instance-specific SW-related data (mappings, files, configurations) are stored into a Kactus2 system design, whereas HW-related data (VHDL, generics, connections to other components) are saved in a Kactus2 HW design. Note that native IP-XACT does not allow an instance-specific definition of SW, which could easily explode the number of CPU components (see Section 2.2.2).

In this use case, the TTA processor designer, the SW programmer, and the SoC designer can be the same person or each part may be implemented by a different engineer or group. Especially in the latter case, comprehensive auxiliary documentation is vital and should be included in the file sets of the components. For example, the processor designer should write a document that describes how to use any custom FUs in C code or include this information in the “description” field of the processor's IP-XACT component.

The design flow is summarized in Table 5 and explained in more detail after it. Many of the phases are similar to the ones in the previous chapter so they are not explained in as much detail here.

Table 5: The design flow phases of a general purpose TTA processor design and integration. Phases G1-G9 contain the processor design and phases G10-G18 the SoC design and implementation on a FPGA.

#	**Phase**	**Input**	**Tools**	**Output**
G1	TTA processor architecture design	Processor specification documents or reused ADF	*ProDe*	Initial processor ADF
G2	Design space exploration	Initial ADF, processor specification documents, test program source codes	*Proxim*, *TTASim*, *ProDe*, *TCECC*, *OSEd*, *Explore*	Final processor ADF
G3	*Optional:* VHDL implementation for FUs if not already in HDB	FU requirements	VHDL writing tools, *HDBEditor*	VHDL files for FUs, augmented HDB
G4	FU implementation mapping	Final ADF, HDB	*ProDe*	IDF file
G5	Estimation of physical properties	Final ADF, IDF, test program source codes	*Estimate*, *TCECC*	Estimation results
	Either G6 if the estimation results are satisfactory or back to G2 if not			
G6	*Either G6.a or G6.b*			
G6.a	Processor generation with *Platform Integrator*	Final ADF, IDF, dummy program TPEF file	*Platform Integrator*	Top-level and hierarchical VHDL files of the processor and memory controllers, top-level entity VHDL
G6.b.1	Manual processor generation	Final ADF, IDF	*ProGe*	Top-level and hierarchical VHDL files of the processor
G6.b.2	Manual memory controller generation	Memory format specifications	VHDL writing tools or target platform's memory controller generator	Memory controllers' VHDL files

#	**Phase**	**Input**	**Tools**	**Output**
G6.b.3	Manual top-level VHDL generation	Top-level VHDL files of processor and memory controllers	VHDL writing tools	Top-level VHDL entity integrating the processor and memory controllers
G7	Documentation writing	Processor specifications, data from previous steps	Document writing tools	Processor's user reference document
G8	*Either G8.a or G8.b*			
G8.a	IP-XACT component creation (if *KoskiIntegrator* was not used in G6)	All the files created in the previous steps	Kactus2 component creation wizard	IP-XACT component of the TTA in the library
G8.b	IP-XACT component augmentation (if *KoskiIntegrator* was used in G6)	IP-XACT file created by *KoskiIntegrator*, all the files created in the previous steps	Kactus2	Augmented IP-XACT component of the TTA in the library
G9	CPU definition	IP-XACT component of the TTA	Kactus2	IP-XACT component of the TTA with CPU definition
G10	SoC HW design	IP-XACT components of the TTA and other components in the design	Kactus2	Kactus2 HW design
G11	SW application design and testing on a PC *or* reuse	Application specification documents *or* reused code	SW code writing tools, testing environment	Platform-independent SW source code
G12	SW source code modification for TTA	Platform-independent SW source code	SW code writing tools	TTA-tailored SW source code files in SW components in the library
G13	SoC system design	Kactus2 HW design, SW IP-XACT components	Kactus2	Kactus2 system design

#	Phase	Input	Tools	Output
G14	Program compilation	Processor ADF, SW source code files	*TCECC*	TPEF files
G15	Program memory image generation	ADF, TPEF files	*PIG*	Instruction and data memory images in specified format
G16	Top-level VHDL generation of the HW design	Kactus2 HW design	Kactus2	Top-level VHDL of the HW design
G17	FPGA synthesis project file generation	Kactus2 HW design	Kactus2 (for Quartus II) or target platform's synthesis tool	FPGA synthesis project files
G18	FPGA synthesis and programming	All the VHDL files, memory image files, FPGA synthesis project files	FPGA synthesis and programming tools	The entire design synthesized and programmed on FPGA

4.1 Processor design

The flow begins now with processor architecture design. The architecture is created with the *ProDe* tool either from scratch or by modifying an existing architecture in the library. Since the processor is intended to run various different applications that are likely unknown to the processor designer, similar performance optimization as was observed in the design flow of Chapter 3 cannot be reproduced. However, there probably exist some general specifications on the cost and performance of the processor which can be used in the designing. The performance can be tested for example by running test programs on the simulated processor. The test programs should be selected according to the intended purpose of the processor. For example, if the processor is targeted for DSP purposes, then a program performing DSP computation should be used. If, on the other hand, it is a true GPP, then several different application domains should be used in the testing. The test programs are compiled for the processor using the *TCECC* tool, and *Proxim* and *TTASim* tools are used for processor simulation yielding data on cycle count and FU utilization. The processor architecture is iteratively modified and tested, possibly with the help of the *Explore* tool, until a satisfactory one is found, and the end result is saved in an ADF file.

Next, a VHDL implementation of the processor is created. Each FU is mapped to a HW implementation in the HDB using the *ProDe* tool. Several different mappings can be created corresponding to different target platforms. For FUs lacking a VHDL imple-

mentation, one must be created and information on them is saved in the HDB using the *HDBEditor* tool. The mappings are saved as IDF files.

With the implementation mappings, some further testing can be done. The TCE tools include *Estimate* that gives estimates on energy consumption, die area and maximum clock rate of TTA designs. The tool takes the processor's ADF and IDF files as input for area and clock rate estimation, and also a sample compiled program file for energy consumption estimation. If the results of estimation do not match the requirements, the architecture can be modified again or more efficient VHDL implementations can be written.

The processor's connected and synthesizable VHDL implementation is created using either *ProDe* or the command-line *ProGe*. The ADF and IDF files are given as parameters, and the output is a hierarchical VHDL description of the processor in target directory. This step is merged with the next one if the *Platform Integrator* tool is used since the integrator creates the connected processor VHDL automatically.

The next step is to create memory controller units for instruction and data memory if they are intended to reside on-chip. These can be handwritten VHDL or the target platform may have a creation wizard for memory controllers. A top-level VHDL connecting the memory controllers to the TTA core is then written. Alternatively, the TCE tools may include a *Platform Integrator* for the target platform that performs these steps automatically. When using the *Platform Integrator*, a SW application in TPEF format is needed as input parameter. In either case, whether using the *Platform Integrator* or not, this phase locks the memory type format which should be mentioned in the processor documentation. The relative directory to which the program memory images of the processor's application should be saved must also be specified in the documentation along with their required file names.

Since the processor can be utilized by a different engineer than who designed it, it is imperative to write documentation on its usage. Besides general information on the processor's performance and cost characteristics, instructions on how to utilize any FUs that are invoked by macros in C code should be included. A list of all the items that should be mentioned in the processor documentation is shown here for reference.

- *Target platform*
- *Targeted purpose if any*
- *Die area*
- *Maximum clock frequency*
- *Special FUs*
- *I/O ports*
- *Macros that are used in C code when utilizing FUs*
- *Relative path of the directory for program image files*
- *Format of the instruction and data memory files*

- *Name of the instruction and data memory files*

The final step in the processor design part of the flow is to create its IP-XACT description. The *KoskiIntegrator* may be used when utilizing the *Platform Integrator*, or alternatively one can use the component creation wizard in Kactus2. The *KoskiIntegrator* does not add any information to the file sets so it must be added in Kactus2 when that option is used. When using the component creation wizard, all the files related to the processor's implementation and documentation are added to the file sets by the user, and its IP-XACT description is otherwise automatically created from the top-level VHDL file. Some or all parts of the processor documentation may also be added to the "description" field of the component. If the processor has multiple implementations for various platforms, then a different file set should be created for each one. They can be further differentiated by creating a separate view for each implementation. The IP-XACT component is then saved to the library with a unique VLNV identifier.

An address space definition is next added to the component in Kactus2 by opening it for manual editing. Dummy values can be used since actual values can be known only when the component is instantiated in a design. This is done in order to be able to add a CPU definition to the component which needs an associated address space. The CPU definition is required for mapping SW components to the TTA component in Kactus2 system designs. After adding the CPU definition, the component is ready in the library.

4.2 SoC design

The processor is now ready to be used in Kactus2 HW and system designs by the SoC designer. The usage is as follows. First, a HW design is created and the TTA components are instantiated and connected in it along with any other components from the library. Next, SW applications, which are reused or written from scratch, are mapped to the TTA instances in a system design with all instance-specific SW-related data saved in the system design to prevent bloating the library. If new SW is created, the corresponding SW components are added to the library before mapping.

If the program source code was reused and not already targeted for the TTA processor on which it is mapped, the SoC designer needs to modify it accordingly. This means inserting macros in the code when a custom FU is used or an external I/O operation is performed using a FU that is not implicitly utilized by a related C language function. For example, the *printf()* function is mapped to a FU operation by default, so this step can be omitted for it. The processor documentation should contain the pertinent user information.

Next, the source code is compiled into TPEF format using *TCECC* and giving the source code files and the processor's ADF file as input parameters. A bit image of the compiled program is then created using *PIG*. The memory image formats are selected according to information in the processor documentation, and output is saved in the

specified directories with the specified file names. Consequently, each processor will run the correct program. The designer then adds the memory image files to the file sets in the system design.

There is one caveat when instantiating the same processor multiple times in the same HW design. By default, they all have the same VHDL implementation including entity and program image names. Thus, if they run different programs, the VHDL needs to be copied and modified so that each processor has a unique entity name and the program images either have different file names or reside in different directories.

The final SoC design can now be saved in the IP-XACT library. Kactus2 creates the top-level VHDL file from the HW design, and also the FPGA synthesis project files if using Altera's Quartus II program for synthesis. If not, the project file can be created using the target platform's corresponding application. All the files needed for FPGA synthesis and programming are now in the library.

5 TTA AS AN ASIP

This use case utilizes the potential of the tool set maximally. Instead of using a given processor from the IP-XACT library, the user generates a custom processor for his C code using a template processor as a starting point architecture.

First, the user connects the selected starting point TTAs from the IP-XACT library into the SoC design using Kactus2. Then, he creates a system design, mapping SW to each TTA processor. Next, he performs an architecture exploration for each TTA processor instance based on the mapped C code. This can be a manual process using the processor simulators in the TCE tools or partially automated with the *Explore* tool.

When using *Explore*, the user defines limits, for example the maximum application run-time, or a minimum clock frequency that provide targets for the exploration. The tool automatically iterates over various TTA architectures, and retargetable compiler and simulator produce cycle-accurate evaluation of the cycle count. The exploration terminates when a maximum number of iterations is reached or upon finding an adequate performance-cost trade-off. At least dozens or hundreds of TTA configurations can be explored within an hour or so. The tool lists a ranking of architecture candidates, from which the user may choose one.

The TTA instances in the SoC design are replaced with the ones found in exploration. Finally, the *TCE Compiler*, *Platform Integrator* and *Program Image Generator* tools are used to generate the files needed in FPGA synthesis of the processors.

The key point here is to keep the number of library items small. There are only a few different template TTAs in the library and more data are stored into the SoC component and a TTA copy linked to it. Otherwise, the amount of views in the template TTAs could grow unmanageably high if there was one for each instance in each SoC design where it is instantiated.

In this use case, the designer considerations are largely the same as in the previous chapter. The HW template creator, the SW designer and the SoC designer can be the same or different individuals. Cooperation between the designers is implicit to the design flow thanks to the documentation and library management capabilities of Kactus2. Compared to the previous design flows, the SoC designer needs more HW expertise, since he must customize and create implementation for the processor template. However, unless custom FUs need to be created, no VHDL expertise is required.

The design flow phases are summarized in Tables 6 and 7, and a more detailed explanation follows them. Since the design flow comprises of many of the same steps and

uses the same tools as in the previous two chapters, some details have been omitted that were presented earlier.

Table 6: The design flow phases of a template TTA processor creation.

#	**Phase**	**Input**	**Tools**	**Output**
T1	Template TTA processor architecture design	Processor specification documents	*ProDe*	Template TTA ADF
T2	Template TTA documentation writing	Processor specification documents, ADF	Document writing tools	Template TTA documentation
T3	IP-XACT component creation for the template TTA	Specified in T3.1-T3.3	Kactus2	Template TTA component in the library
T3.1	Bus interface creation	Processor specification documents	Kactus2	Bus interface definitions in the component
T3.2	File set creation	Template TTA ADF, documentation	Kactus2	File sets in the component
T3.3	Address space creation and CPU definition	None needed	Kactus2	Address space and CPU definitions in the component

Table 7: The design flow phases of a custom processor creation and integration.

#	**Phase**	**Input**	**Tools**	**Output**
A1	SW application design and testing on a PC *or* reuse	Application specification documents *or* reused code	SW code writing tools, testing environment	Platform-independent SW source code components in the library
A2	SoC-specific TTA component creation	Template TTA in the library	Kactus2	SoC-specific copy of the template TTA in the library

#	**Phase**	**Input**	**Tools**	**Output**
A3	SoC HW design creation	IP-XACT components of the TTA and other components in the design	Kactus2	Kactus2 HW design
A4	TTA component view creation	SoC-specific TTA component	Kactus2	SoC-specific TTA component with instance-related views
A5	SoC system design creation	Kactus2 HW design, SW components	Kactus2	System design view in the HW component
A6	C code modification for TTA	Platform-independent SW source code, template TTA documentation	SW code writing tools	TTA-tailored source code files
A7	TTA architecture optimization for application SW	Template ADF, TTA-tailored source code files	*Proxim*, *TTASim*, *ProDe*, *TCECC*, *OSEd*, *Explore*	Final TTA ADF files, final TPEF files
A8	*Optional:* VHDL implementation for FUs if not already in HDB	FU requirements	VHDL writing tools, *HDBEditor*	VHDL files for FUs, augmented HDB
A9	FU implementation mapping	Final TTA ADF files, HDB	*ProDe*	IDF files
A10	Estimation of physical properties	Final ADF, IDF, final TPEF	*Estimate*	Estimation results
	Either A11 if the estimation results are satisfactory or back to A7 if not			
A11	*Either A11.a or A11.b*			
A11.a	Processor generation with *Platform Integrator*	ADF, IDF, TPEF files	*Platform Integrator*	Top-level and hierarchical VHDL files of the processors and memory controllers, top-level VHDL entity
A11.b.1	Manual processor generation	ADF, IDF files	*ProGe*	Top-level and hierarchical VHDL files of the processors

#	Phase	Input	Tools	Output
A11.b.2	Manual memory controller generation	Memory format specifications	VHDL writing tools or target platform's memory controller generator	Memory controllers' VHDL files
A11.b.3	Manual top-level VHDL generation	Top-level VHDL files of the processors and memory controllers	VHDL writing tools	Top-level VHDL files integrating the processors and memory controllers
A12	Program memory image generation	ADF, TPEF files	*PIG*	Instruction and data memory images in specified format
A13	Top-level VHDL generation of the HW design	Kactus2 HW design	Kactus2	Top-level VHDL of the HW design
A14	FPGA synthesis project file generation	Kactus2 HW design	Kactus2 (for Quartus II) or target platform's synthesis program	FPGA synthesis project files
A15	FPGA synthesis and programming	All the VHDL files, memory image files, FPGA synthesis project files	FPGA synthesis and programming tools	The entire design synthesized and programmed on FPGA

5.1 Template processor design

In this use case, one or more starting point architectures are created in the IP-XACT library. This is only necessary once and is not considered to be part of the integration flow. Different architectures can be created with *ProDe* and documentation on their properties, limitations and intended usage should be written. For each architecture, a template IP-XACT component is created using the component creation wizard in Kactus2. This allows their integration into HW designs even though they don't have VHDL implementation yet.

After creating the IP-XACT components, they are further edited in Kactus2. For each component, bus interfaces need to be defined to allow connection to other components in Kactus2 HW designs. This includes adding bus definitions and mapping them to abstraction definitions from the library. File sets are also created by the user but they

contain only the ADF file and documentation at this point. Alternatively, documentation can be fitted in the “description” field of the component. Each component also requires an address space definition. This is necessary so that Kactus2 understands that it is a CPU component that is visible in a system design. Address space depends on the design where the component is instantiated so dummy values can be used here. Finally, the user defines a CPU for the component and references the address space in it. The component is then saved in the library for use and customization as an ASIP.

5.2 ASIP and SoC design

The integration flow begins with SW design. The application can be reused or newly written. It should be tested for correct functionality on a workstation PC before retargeting for TTA. Each unique application SW package is saved in the IP-XACT library as a SW component with the source code files in the file sets.

Next, a HW design is created with Kactus2. The TTA template component saved in the library is not itself instantiated in the design. Instead, a copy of it is created and that copy is used in the design, and in that design only. The reason for this is tied to library management. If the original component was used, it could eventually contain an unwieldy amount of views, one for each instance in each design where it was used. On the other hand, if a new component was created for each instance, the library would explode with copies of the component. The solution is thus to create one copy for each design where it is used, and create instance-specific views in the copies.

After creating the copy, the designer instantiates and connects the copied template TTAs into his HW design along with other components and saves the design in the IP-XACT library as a new component. For each instance of the TTA component in the design, a new view is created for it in Kactus2. These views are later linked to file sets referring to the instance's implementation. In the HW design, the user links instances to their respective views. A system design mapping the SW applications to TTA instances in the HW design is then created with Kactus2 and saved in the library as a system view of the component generated in the previous step.

Since the TTAs may contain custom FUs that need to be invoked with specific macros, the application source code must next be modified accordingly. The processor documentation should contain information about the presence of these FUs and on their usage with macros. After source code modification, it is compiled for the corresponding TTAs using *TCECC*.

Now, that the processors have applications mapped to them, their architecture can be explored for efficiently running the programs. This can be done using *Proxim* or *TTASim* as in the previous chapters and the *Explore* tool can be used to help as described in the beginning of this chapter. The SW source code is modified and compiled for the final architectures, and the SW components' file sets are augmented with these

files in the library. New views referencing the TTA-specific file sets should also be created in the SW components so that they can be linked to the SW instances in the system design. Any new custom FUs created are added to the HDB.

After finding efficient architectures for the processors, their VHDL implementation is generated using the *ProDe* and *ProGe* tools. The FUs of each processor are mapped to their implementations in the HDB and the mappings are saved as IDF files. The *Estimate* tool is next used to estimate the physical properties of the processors, and design space exploration is resumed if the results are not good enough. The memory controllers and the top-level VHDL files are created using the *Platform Integrator* tool if it supports the target platform. Otherwise, the memory controllers and the top-level VHDL of the TTAs must be created manually. The implementation files are saved in the file sets of the TTA component created for the design. Each instance has its own file set where the corresponding files are included.

Next, the memory images of the SW applications are created using the *PIG* tool. This tool is invoked for each processor-application pair. The created files are saved in the file sets corresponding to the processor instance on which they are used.

The top-level VHDL file of the HW design can now be created using the Kactus2 VHDL generation tool along with Quartus II project files, if applicable, using the Kactus2 project file generator. The design is now ready to be synthesized and programmed on a FPGA.

6 AN ILLUSTRATIVE EXAMPLE WITH CRC

In this chapter, a use case of the design flow described in Chapter 5 (ASIP) is presented to illustrate the details one encounters when working with the TCE tools and Kactus2 within the framework of the flows presented in this Thesis. The design presented is rather simple to keep focus on the flow instead of the intricacies of the design. A simple TTA processor template with custom FUs enabling real-time clock and printing to standard output is first created in the IP-XACT library. The processor template is then used for an application that counts a 32-bit cyclic redundancy check (CRC) [27] for a block of data and prints the result to the standard output. Two processor implementations are created: One is used as such from the template in the library and the other is modified to run the application faster. Both implementations are instantiated in a SoC design for synthesis on a FPGA.

6.1 Template component creation

First, the template TTA processor is created using *ProDe*. Figure 15 shows the architecture as presented in *ProDe*. The processor contains only a load/store unit (LSU), an arithmetic-logical unit (ALU), two register files (RFs), a global control unit (GCU), a FU for real-time clock (rtc_timer) and a FU that can print to the standard output (stdout) of the platform. There is only one transport bus and the processor is fully connected. The architecture is saved in an ADF file. This processor template can be used for any application that requires timer and printing capabilities.

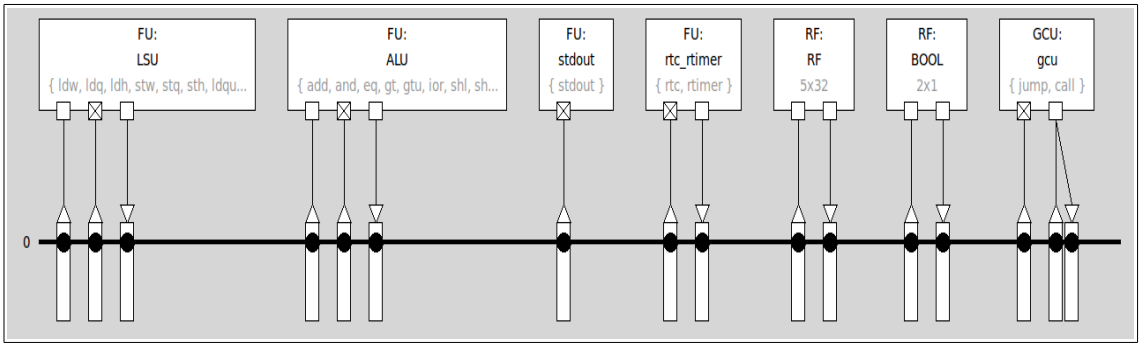

Figure 15: The template processor architecture as seen in ProDe. *Input ports with "X" denote ports that trigger the FU.*

Next, the processor template is saved in the IP-XACT library using Kactus2. This is done with the component creation wizard. A unique VLNV identifying the component is given (in this case: SL/ip.cpu/TTA_template/1.0). The IP-XACT file of the component is now in the library under the folder indicated by the VLNV values, but it requires further editing. First, its external ports and bus interfaces are defined in Kactus2. The processor has only two ports: clock and reset. No explicit port for the standard output is needed thanks to the implementation that will be selected for the stdout FU which utilizes Altera's JTAG UART [28, Chapter 5] to send the output to PC.

The port data is entered in the opened component's port editor as shown in Figure 16. A bus interface is created for both ports. In this case, standard bus definitions and abstraction definitions for clock and reset ports already exist in the library so they don't have to be created. The bus interface editor with the values for the clock bus is shown in Figure 17. The physical clock port is mapped to the bus interface in the “Port maps” tab.

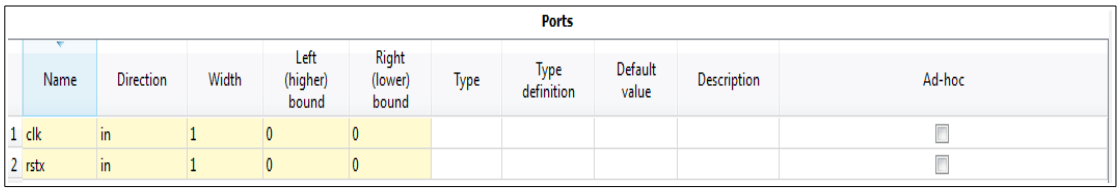

Figure 16: The values given in the ports editor in Kactus2. Only the mandatory values have been entered for these simple ports.

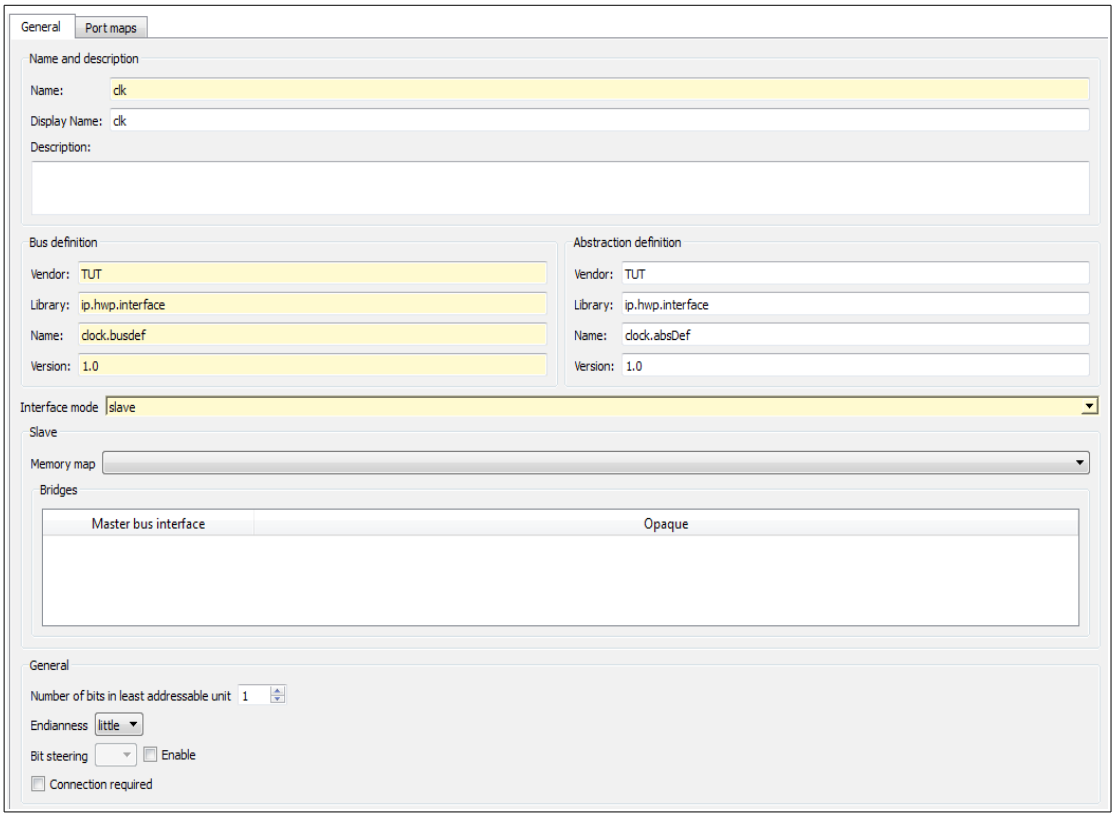

Figure 17: The values given in the bus interface editor for the clock bus in Kactus2.

The file sets editor is opened next where a single file set “TCEFiles” is created, and the ADF file is added to the file set. Then, an address space is created in the corresponding editor and it is mapped to a CPU in the CPU editor, so that Kactus2 knows that this is a CPU component. Since this TTA processor doesn't control any external components, the address space can be left empty. Finally, a short description of the processor is written in the component's description field for user reference. For a more complex processor, a documentation file should be written and added to the file sets. The component is now ready for use in the library.

Figure 18 shows the finished TTA template component view in Kactus2. The entries in bold font on the left contain information that was added to the component.

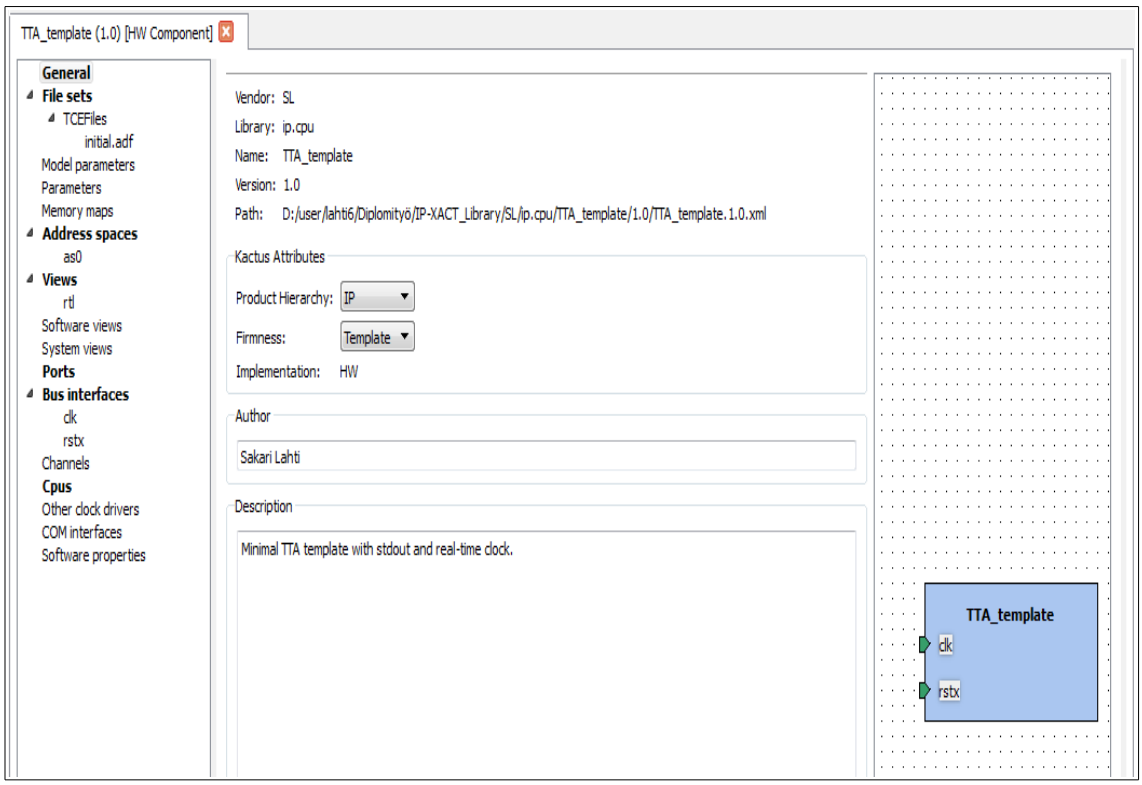

Figure 18: The component view of the finished TTA template component.

6.2 Using the component in a SoC design

The application is a C program that counts a 32-bit CRC for a block of data and prints the result to the standard output. The source code contains three files: “main.c” for a simple main program, “crc.c” that contains the function calculating the CRC value and “crc.dat” that contains a precalculated look-up table for making the algorithm faster. The contents of the source code files themselves are not relevant to this work. The correct functionality of the program has been tested on a workstation PC prior to reuse in a

TTA, and it has been saved to the library as a SW component (SL/sw/crc/1.0) with the source code files in the file sets.

The template TTA component is not used itself in the SoC design for reasons described in Chapter 5, so a copy of it is created for instantiation. This copy is given a new VLNV (SL/ip.cpu/TTA_CRC/1.0). A new HW design is then created in Kactus2 (SL/soc/crc/1.0) where the new component is instantiated. In this case, it is used twice since the intent is to compare the performance of a simple and a more complex custom processor in counting the CRC. The external ports for clock and reset signals are created and connected to the TTA components. The library and design now look like in Figure 19.

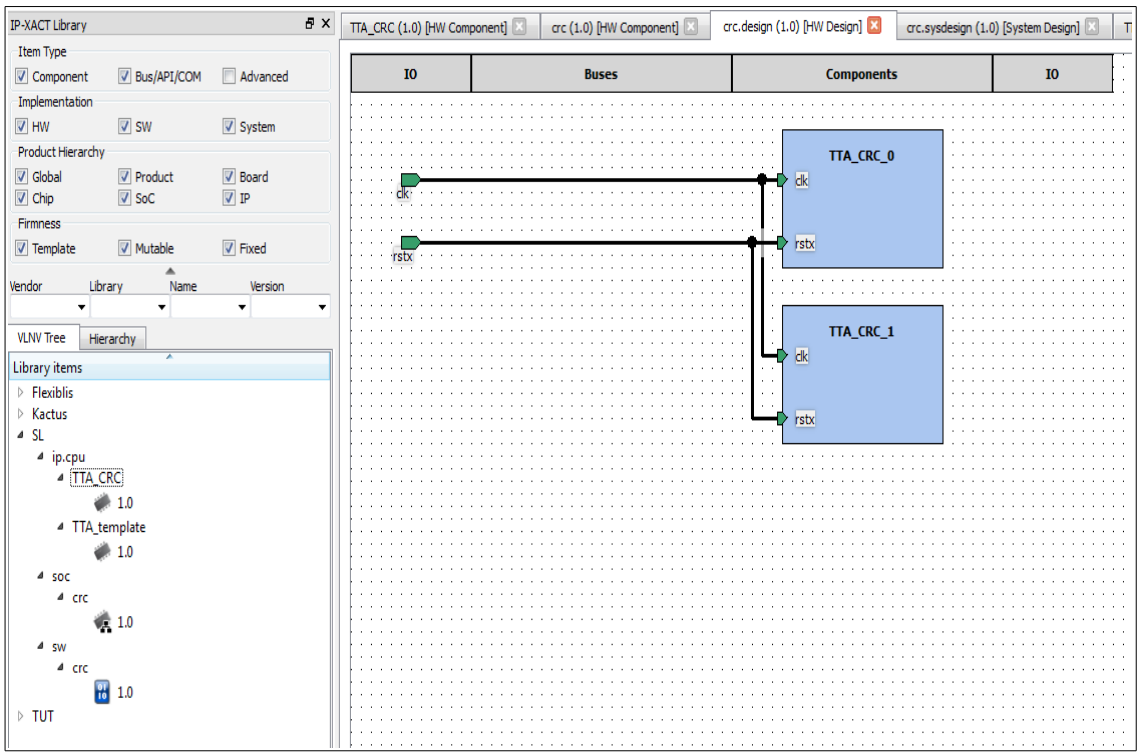

Figure 19: The library (in the left) and the HW design with two TTA instances.

Since the TTA component is instantiated twice in the HW design, two different views are created for it using the views editor in Kactus2. The views differ in name, description, model name and file set references. The views editor for the simple TTA core is shown in Figure 20. Each TTA's file set is also added at this point to the component so that they can be referenced in the views. The model names will later be used as the entity names when generating top-level VHDL files for the processor cores. In the Kactus2 HW design, the instance names are linked to the corresponding views.

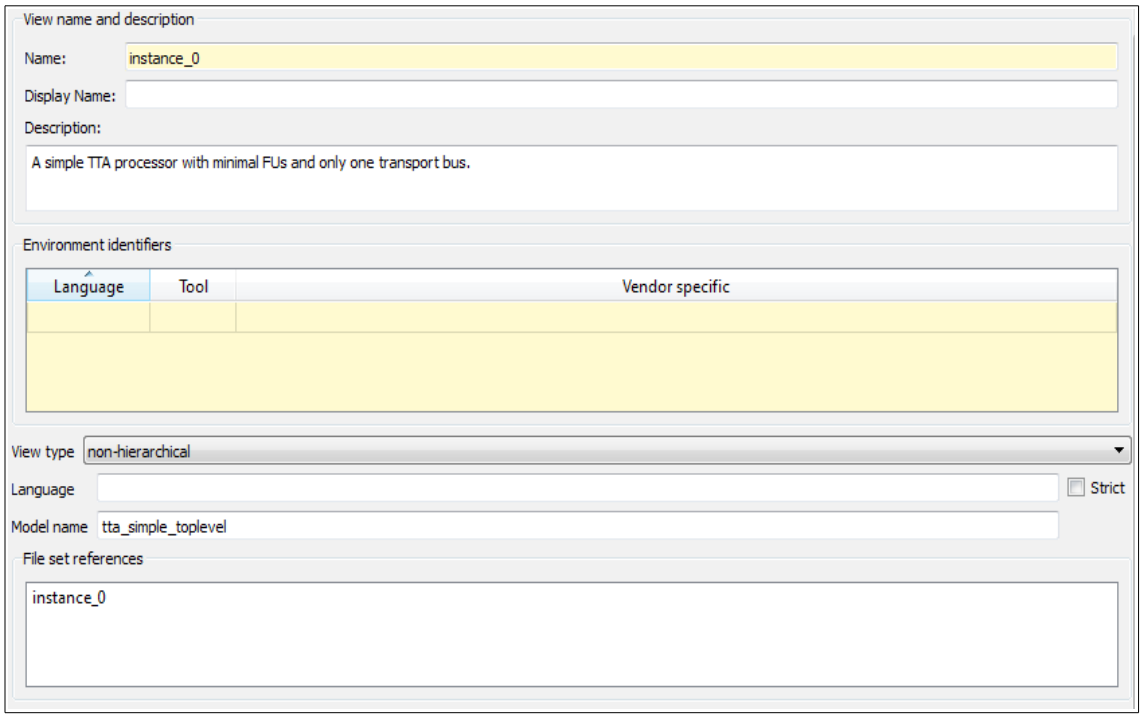

Figure 20: The Kactus2 views editor for the simple TTA core.

The system design mapping the SW applications to the instantiated processors is created next using the Kactus2 component creation wizard. In this case, the system design is simple and shown in Figure 21. Both processors run the same application and there are no inter-processor communication channels. The system design is not saved as a new component but as a system view to the HW design component. This keeps the library compact and it is clear which HW design the system design is linked to. Together they form a complete SoC design description.

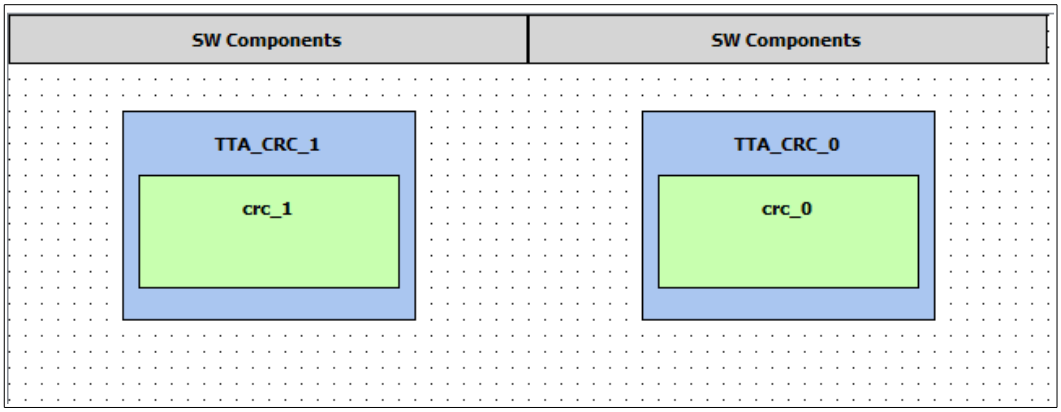

Figure 21: The system design mapping SW to TTAs. The blue boxes represent the TTAs and the green boxes the SW.

Next, the application source code is modified for running on a TTA processor. In this case, there is the real-time clock FU that is invoked with specific macros. Macros for

starting and stopping the clock and yielding the elapsed time are added to the program. Furthermore, the standard *printf()* function is replaced with an *iprintf()* function that drops support for printing floating point values but is much faster on a TTA. The program is then compiled for TTA using the *TCECC* tool resulting in a TPEF file. The modified source code files and the TPEF file are saved in the SW component's file sets, with a new file set being created for the TTA-tailored program. A new view referencing this file set is also created for the SW component. In the system design, the created view is selected for the SW component instances.

In this example, one of the processors is customized to run the application more efficiently. The design space exploration for this is outside the scope of this work but its description can be found in [21, Section 3.1]. The end architecture is shown in Figure 22. The differences to the initial architecture include a new custom FU (REFLECTER) that performs a reflection operation for input bit-pattern needed by the CRC algorithm, and three additional transport buses allowing instruction-level parallelism [29, Chapter 2]. A VHDL implementation for the reflecter FU is also written and saved in the HDB. The source code is modified and compiled to use the new custom FU, but the unmodified source code and TPEF file are also retained since they are used by the simpler TTA instance.

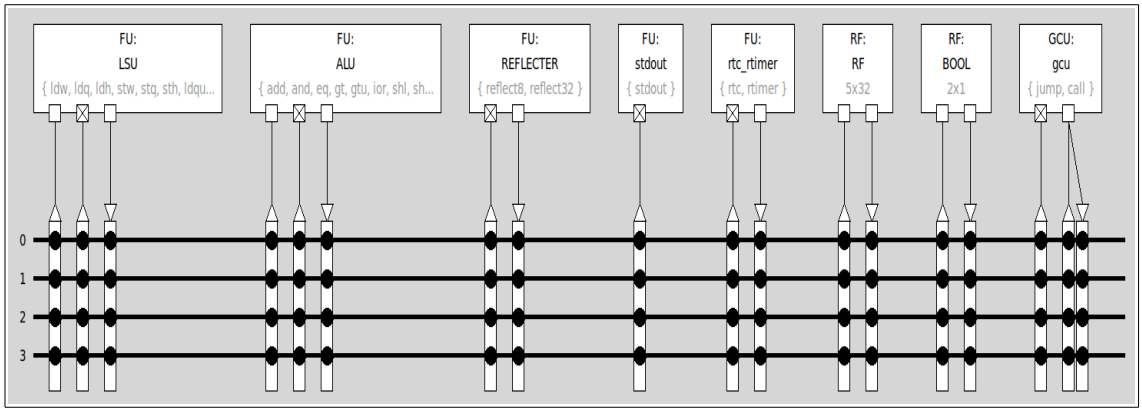

Figure 22: The architecture of the TTA customized for CRC calculation.

The TTA cores need a VHDL implementation which is created next. First, implementations are selected for the FUs using *ProDe*. At this point, the target platform must be known since each platform may have a different implementation for certain FUs. Once each FU is mapped to an implementation from the HDB, the mapping is saved as an IDF file. The VHDL is now ready to be generated. In this case, the target platform is Altera Stratix II DSP Pro board which has *Platform Integrator* support. For both processors, the corresponding ADF, IDF and TPEF files are given as parameters to the *ProGe*. The model names that were used in the instance-specific views are passed as parameters to be names of the top-level VHDL entities. Other given parameters include the instruction and data memory types. The output from the generator contains the VHDL files im-

plementing the processor cores, memory controller units and top-level entities linking them together. These are saved to the file sets of the TTA component in the library with a separate file set for each processor instance in the design.

The program images are generated for each processor using *PIG,* with the ADF and TPEF files given as parameters. The generated instruction and data memory image files are saved in the file sets corresponding to their TTA instances. The design-specific TTA component in the library is now complete.

The top-level VHDL file of the HW design instantiating the processors is automatically generated by invoking the Kactus2 tool purposed for this task. It simply instantiates the processors and connects them to the external ports. The processor instances reference the model names in the instance-specific views so that they are linked to the correct VHDL implementations. This VHDL file is saved to the file sets of the HW design. The Quartus II project files can now be created using the generator tool in Kactus2, following which FPGA synthesis and programming is straightforward. When run on FPGA, the time each processor used for the CRC calculation could be printed on Altera's Nios2-terminal tool running on a workstation PC connected to the FPGA's JTAG UART port.

7 DESIGN FLOW EVALUATION

While making this Thesis, several Kactus2 and TCE use cases were carried out. Based on them, an estimation of the time taken by each phase in the design flows has been formed. The estimated times assume a reasonably experienced user who is familiar with the tools. Both Kactus2 and the TCE tools can be executed on Linux on a standard PC computer or alternatively Kactus2 can be run on Windows and the TCE tools on a Linux virtual machine as the author did. Setting up both tools is straightforward and is counted as a non-recurring task.

7.1 TTA as a fixed accelerator IP block

Time-usage evaluation of the use case where TTA is treated as a fixed accelerator IP block (Chapter 3) is shown in Table 8. The time taken by SW design from scratch (phase F1) is not dependent on the design flow and greatly varies by the application so it is not counted against the time used in the flow. However, if SW is reused from the library, little time is consumed.

Table 8: Time-usage estimations of each phase in the design flow of TTA as a fixed accelerator IP block (as in Table 4).

#	**Phase**	**Time**
F1	SW application design and testing on a PC *or* reuse	Dependent on application *or* < 5 min
F2	Initial TTA processor design	15 min
F3	Program compilation for initial TTA	< 5 min
F4	Design space exploration	20 min to few hours
F5	C code modification and compilation for TTA	< 5 min to 1 h depending on the size of source code and the amount of custom FU usage
F6	*Optional:* VHDL implementation for FUs if not already in HDB	30 min to few hours per FU
F7	FU implementation mapping	5 min

#	**Phase**	**Time**
F8	Estimation of physical properties	10 min
	Either F9 if the estimation results are satisfactory or back to F4 if not	N/A
F9	*Either F9.a or F9.b*	N/A
F9.a	Processor, memory controller, and top-level entity generation with *Platform Integrator*	< 5 min
F9.b.1	Manual processor generation	< 5 min
F9.b.2	Manual memory controller generation	10 min to 1 h
F9.b.3	Manual top-level entity VHDL generation	20 min
F10	Memory image generation	< 5 min
F11	*Either F11.a or F11.b*	N/A
F11.a	IP-XACT component creation (if *KoskiIntegrator* was not used in F9)	10 min
F11.b	IP-XACT component augmentation (if *KoskiIntegrator* was used in F9)	10 min
F12	SoC design	15 min + 2 min/component
F13	Top-level VHDL generation of the HW design	1 min
F14	FPGA synthesis project file generation	1 min with Kactus2 or 10 min with platform-specific program
F15	FPGA synthesis and programming	Dependent on the target platform and size of the design
Minimum total time		Approx. 1 h 40 min + 2 min/component

Designing the initial processor is fast in *ProDe* as is compiling the program for the architecture (phases F2 and F3). Design space exploration (phase F4), on the other hand, can take more or less time depending on how strict the performance requirements are and how many FUs and transport buses can be added to the TTA. Moreover, designing custom operations and implementing them in program code can take hours. C code modification for TTA (phase F5) takes from a few minutes to an hour based on the amount of source code and custom FU usage. In the best case, with no custom FUs, this phase can be skipped. If new custom FUs were created during design space exploration,

they need VHDL implementation (phase F6) which can take anywhere from half an hour to several hours to write for each custom FU.

Phases F7 to F11 consist of creating the processor IP block from the source files generated in the previous steps. Thanks to the automated tools, this takes only about half an hour with *Platform Integrator* support or two to three times more if it cannot be used.

Counted together, phases F1 to F11, which consist the TTA IP block creation, take little more than one hour in the best case scenario where the SW application is ready in the library, no custom FUs are added in the design space exploration and *Platform Integrator* can be used. At most, the entire design flow can take a few work days if extensive design space exploration is performed with various custom FUs created to speed up the processor. This is still fast when compared to an approach where the whole IP block is created manually in VHDL without using SW running on a processor. On the other hand, while using a GPP would take less time, it would suffer from the problems of using more die area, performing worse and consuming more energy.

The integration of the TTA IP block in SoC designs is effortless using Kactus2 (phases F12-F15). It should take less than an hour even for relatively large designs assuming that everything is ready in the library. FPGA synthesis and programming can take anywhere from few minutes to hours depending on the target platform and size of the design.

7.2 TTA as a general purpose processor

Table 9 provides the estimation of time used in the design flow where TTA is used as a GPP (Chapter 4). Many of the phases and their time consumption are the same as in the previous case. Documentation writing (phase G7) is a new phase that takes from one to few hours depending on how formal the document has to be and whether there is a document template available. In the SoC design part of the flow, most phases are repeated for each processor instance in the design, so the time usage has a linear correlation to the amount of processor instances. The time spent for the system design (phase G13) additionally depends on whether MCAPI channels, which require some configuration, are used.

Table 9: Time-usage estimations of each phase in the design flow of TTA as a general purpose processor (as in Table 5).

#	**Phase**	**Time**
G1	TTA processor architecture design	15 min
G2	Design space exploration	20 min to few hours

#	**Phase**	**Time**
G3	*Optional:* VHDL implementation for FUs if not already in HDB	30 min to few hours per FU
G4	FU implementation mapping	5 min
G5	Estimation of physical properties	10 min
	Either G6 if the estimation results are satisfactory or back to G2 if not	N/A
G6	*Either G6.a or G6.b*	N/A
G6.a	Processor generation with *Platform Integrator*	< 5 min
G6.b.1	Manual processor generation	< 5 min
G6.b.2	Manual memory controller generation	10 min to 1 h
G6.b.3	Manual top-level VHDL generation	20 min
G7	Documentation writing	1 to 3 hours
G8	*Either G8.a or G8.b*	N/A
G8.a	IP-XACT component creation (if *KoskiIntegrator* was not used in G6)	10 min
G8.b	IP-XACT component augmentation (if *KoskiIntegrator* was used in G6)	10 min
G9	CPU definition	< 5 min
G10	SoC HW design	15 min + 2 min/component
G11	SW application design and testing on a PC *or* reuse	Dependent on application *or* < 5 min per processor
G12	SW source code modification for TTA	< 5 min to 1 h per processor depending on the size of source code and the amount of custom FU usage
G13	SoC system design	1 to 10 min per processor depending on the complexity of the SW stack and MCAPI channels
G14	Program compilation	< 5 min per processor
G15	Program memory image generation	< 5 min per processor
G16	Top-level VHDL generation of the HW design	1 min

#	Phase	Time
G17	FPGA synthesis project file generation	1 min with Kactus2 or 10 min with a platform-specific program
G18	FPGA synthesis and programming	Dependent on the target platform and size of the design
Minimum total time		Approx. 2 h 30 min + 2 min/component + 20 min/processor

All in all, the processor design (phases G1 to G9) can take a couple of hours for a simple processor with no design space exploration, or closer to a day for a complex processor with many newly-implemented custom FUs. SoC design (phases G10 to G18) takes less than an hour for a small design with one processor, or a few hours at maximum for a large system with many processors and SW applications. Again, SW implementation is not counted against the time taken by the design flow.

Compared to designing the system without processors, from ASIC principles, the time saved is tremendous. For more complex applications, it is not even realistic to consider an ASIC approach.

A comparison to using a more traditional GPP is more appropriate. The benefits here are that the library can contain dozens of processors directed for various application domains which allows selecting an efficient processor for the application at hand. The processors are also easy to design in-house which avoids any licensing fees.

7.3 TTA as an ASIP

Tables 10 and 11 provide the estimation of time used in the design flow where TTA is used as an ASIP (Chapter 5). Table 10 shows the time usage of the template processor generation and Table 11 of its customization and integration to a SoC design. Many of the phases are the same as in the previous design flows with similar time usage.

Table 10: Time-usage estimations of each phase in the design flow of a template TTA processor creation (as in Table 6).

#	Phase	Time
T1	Template TTA processor architecture design	15 min
T2	Template TTA documentation writing	15 min to 1 h
T3	IP-XACT component creation for the template TTA	< 5 min

#	Phase	Time
T3.1	Bus interface creation	5 min/bus
T3.2	File set creation	< 5 min
T3.3	Address space creation and CPU definition	< 5 min
Minimum total time		Approx. 40 min + 5 min/bus

Table 11: Time-usage estimations of each phase in the design flow of a custom TTA processor creation and integration (as in Table 7).

#	Phase	Time
A1	SW application design and testing on a PC *or* reuse	Dependent on application *or* < 5 min per processor
A2	SoC-specific TTA component creation	< 5 min
A3	SoC HW design creation	15 min + 2 min/component
A4	TTA component view creation	1 min/processor instance
A5	SoC system design creation	1 to 10 min per processor instance depending on the complexity of SW stack and MCAPI channels
A6	C code modification for TTA	< 5 min to 1 h per processor instance depending on the size of source code and the amount of custom FU usage
A7	TTA architecture optimization for application SW	20 min to few hours per processor configuration
A8	*Optional:* VHDL implementation for FUs if not already in HDB	30 min to few hours per FU
A9	FU implementation mapping	5 min/processor configuration
A10	Estimation of physical properties	10 min/processor configuration
	Either A11 if the estimation results are satisfactory or back to A7 if not	N/A
A11	*Either A11.a or A11.b*	N/A
A11.a	Processor generation with *Platform Integrator*	< 5 min

#	**Phase**	**Time**
A11.b.1	Manual processor generation	< 5 min
A11.b.2	Manual memory controller generation	10 min to 1 h
A11.b.3	Manual top-level VHDL generation	20 min
A12	Program memory image generation	< 5 min per processor instance
A13	Top-level VHDL generation of the HW design	1 min
A14	FPGA synthesis project file generation	1 min with Kactus2 or 10 min with a platform-specific program
A15	FPGA synthesis and programming	Dependent on target platform and size of the design
Minimum total time		Approx. 30 min + 2 min/component + 15 min/processor instance + 35 min/processor configuration

Creating a template processor takes little time, on the order of one hour. Thus, many different processor templates can be quickly created and saved in the library.

The time taken by SoC designing using template TTAs depends on the number of TTA instances and on whether they use the same or different final processor architecture and FU implementations. A design with only one processor with no new custom FUs created for it takes little more than one hour. A complex design with multiple processors and custom configurations could take tens of hours to complete.

Comparison to other implementation methods is quite similar as in the GPP design flow. An ASIC approach would probably be too difficult and time-consuming to attempt with any but the simplest of applications. Against a traditional GPP processor implementation, this method provides processors that are better suited to the target applications and are free of licensing fees. Compared to the previous use case, the ASIP approach puts more design burden on the shoulders of the SoC designer since he creates the processors' internal implementation as well, but the end result is better optimized for the applications.

8 CONCLUSIONS

8.1 General remarks

This work presented three new design flows that integrate the use of the TCE tools and Kactus2 when creating MP-SoC FPGA designs. In the first case, a TTA was designed as a fixed accelerator IP block, running predefined program code. In the second case, the TTA was designed as a GPP with no pre-mapped SW, and in the third case as an ASIP where the final processor architecture and program code are decided during SoC integration. In all the cases, the design flows also showed how to integrate the TTA in SoC designs using Kactus2. All the design flows utilize between 8 and 12 different program tools from Kactus2 and the TCE tools depending on target platform and whether custom FUs are created.

Table 12 contains the summary of the relevant aspects from each design flow. The fixed accelerator IP block option provides the best performance but least customizability, whereas a GPP can be reused for a wide range of applications but offers the worst performance and highest area and energy cost. ASIP stands between these two options in terms of customizability, performance and cost.

Table 12: Summary of the properties of the design flows.

	Fixed accelerator	**GPP**	**ASIP**
Purpose	To create a fixed accelerator IP block in the IP-XACT library and reuse it in SoC designs.	To create a soft-core GPP in the IP-XACT library that can run SW applications and reuse it in SoC designs.	To create a processor template in the IP-XACT library that can be tailored for SW applications and reuse it in SoC designs.
Example	Fixed IP block to calculate a 32-bit CRC	GPP with standard output and streaming I/O capabilities, but without support for interrupts.	ASIP for efficiently running video encoding applications
Customizability	None	SW	SW and HW

	Fixed accelerator	GPP	ASIP
Number of phases in the design flow	15	18	3 for creating a template TTA + 15 for customization and integration
Approx. minimum design and integration time	1 h 40 min + 2 min/component	2 h 30 min + 2 min/component + 20 min/processor	30 min + 2 min/component + 15 min/processor instance + 35 min/processor configuration
Pros	Simple to design and reuse, very efficient for its task	Simple to design and reuse, usable for wide range of applications	Simpler to design than ASIC, efficient for its task
Cons	No customizability, performs only one task	Weaker performance, higher area cost and energy consumption than with the other options	Not as widely usable as a GPP, reuse requires some HW expertise

The presented design flows shorten design time, decrease error-prone tasks, and offer product data management capabilities for a product's entire life cycle. For example, without an integrated design flow the SW code running on a TTA would be "just files on disk," whereas now they can be stored in the IP-XACT library, described with metadata, and referred uniquely from other designs. This simplifies creating derived products later on. Designers can easily browse older designs, check the used versions and parameter values, see where a certain component has been instantiated, and auto-generate combined documentation throughout the whole product including SW, HW and mapping information.

When compared to other design methods, several advantages were found. Against pure ASIC implemented in HDL, there is a significant reduction in design time. Furthermore, for more complex applications, an ASIC approach is not even feasible in most cases. On the other hand, it was found that while using a traditional GPP would save somewhat in design effort, the TTA approach allows for a more efficient design with improved performance and reduced die area and energy consumption, while avoiding licensing fees. The methods described in this work thus offer a good middle-ground between the ASIC and GPP approaches, manifesting advantages from both.

8.2 Suggestions for future development of the tools

The TCE tools and Kactus2 have been developed separately within different research groups and their integrated use has not been a major consideration in either group. The *KoskiIntegrator* is the only notable tool which has been designed with the integrated use in mind, but even it has not been kept up to date as Kactus2 has added extensions to the IP-XACT standard.

The presented design flows would greatly benefit from new properties in both the TCE tools and Kactus2 as shown below. It is estimated that 10% to 20% cuts to the time taken by going through the design flows could be achieved with such new features in the tools, and more importantly, the added automation would reduce the risk of errors in keeping files referenced to the correct designs. Indeed, the file juggling between the two tool sets is currently the greatest hindrance in the efficient use of the design flows. The suggestions below aim to provide ways to fix this.

8.2.1 Suggested modifications to the TCE tools

The *KoskiIntegrator* is the existing bridge between the TCE tools and Kactus2. However, there are a couple of limitations to it. First, it does not create file set information. A file set containing the ADF and IDF files and the VHDL files of the processor should be added to the IP-XACT description. Second, the *KoskiIntegrator* requires a HIBI compatible load/store unit in the processor. This seems like an unnecessary restriction and should be removed.

The *Platform Integrator* would benefit from support to more platforms. Of course, it is infeasible to add support for every platform in existence, so a good solution would be a configurable *Platform Integrator* where one could enter platform-dependent key parameters with a graphical user interface.

8.2.2 Suggested modifications to Kactus2

As it is, the TCE tools and Kactus2 are completely separate program suites and no interoperability is possible between them. Thus, files must be handled in the two tool sets in turns which can be cumbersome and prone to errors. The suggestion is that most TCE tools could be invoked directly from Kactus2 in the following manner.

- The *TCECC* compiler could be invoked from a Kactus2 system design. The system design maps SW on processor instances, so it would be natural to invoke the compiler there for a selected mapping. The compiler uses the SW source code files and the processor's ADF file which can be found in the corresponding file sets. The user would enter compilation parameters in Kactus2 as well.
- *ProDe* could be launched from Kactus2 by selecting an ADF file in the library. The chosen ADF file would be opened in *ProDe*. After launching *ProDe* in this way, any generated IDF file could be added to the file set of the ADF file by user selection.
- *ProGe* and *Platform Integrator* could be invoked from Kactus2. The user would select the ADF file and a corresponding IDF file, enter parameters and launch *ProGe* which would run on background and automatically add the generated implementation files to a file set defined by the user.

- *PIG* could be invoked from a Kactus2 system design by selecting a processor instance with mapped SW components. Kactus2 would automatically give the corresponding ADF and TPEF files to the *PIG* tool and the user would enter other parameters from Kactus2. The created program image files would be added to the specified file set.

With these additions to Kactus2, there would be little need to explicitly invoke the TCE tools by the user and most of the design flow could be operated from Kactus2 environment. The notable exception are the design space exploration tools which are often used in an iterative manner and involve several different programs from the TCE tools, and thus are more naturally used outside the context of Kactus2.

When implementing these features to Kactus2, special care should be placed on considering file sets and views and how they relate to different TTA instances. Otherwise, wrong files could be involved when invoking the tools. It should also be noted that Kactus2 must be run on Linux when invoking the TCE tools from it since the TCE tools do not support the Windows operating system.

REFERENCES

[1] International technology road map for semiconductors [WWW]. 2011 edition, system drivers. [accessed on 11.5.2014]. Available at: http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[2] Rasmus, A., Kulmala, A., Salminen, E. & Hämäläinen, T.D. IP Integration overhead analysis in system-on-chip video encoder. 2007 IEEE workshop on design and diagnostics of electronic circuits and systems (DDECS), Krakow, Poland, April 11-13, 2007. Gliwice, Poland 2007, INTERPRINT s.c. pp. 333-336.

[3] Salminen, E. On design and comparison of on-chip-networks. Ph. D. Dissertation. Tampere 2010. Tampere University of Technology, Publication 872. 230 p.

[4] Snapdragon 800 processors [WWW]. Qualcomm, Inc. [accessed on 11.5.2014]. Available at: http://www.qualcomm.com/snapdragon/processors/800.

[5] Nios II Processor [WWW]. Altera Corp. [accessed on 11.5.2014]. Available at: http://www.altera.com/devices/processor/nios2/ni2-index.html.

[6] Corporaal, H. Microprocessor architectures: From VLIW to TTA. Chichester, England 1998, John Wiley & Sons. 407 p.

[7] Jääskeläinen, P., Guzma, V., Cilio, A. & Takala, J. Codesign toolset for application-specific instruction-set processors. Multimedia on Mobile Devices, San Jose, California, USA, January 29-30, 2007. pp. 65070X-1 - 10.

[8] Jones, A.K., Hoare, R., Kusic, D., Fazekas, J. & Foster, J. An FPGA-based VLIW processor with custom hardware execution. 13Th international symposium on field-programmable gate arrays, Monterey, CA, USA, February 20-22, 2005. New York, NY, USA 2005, ACM. pp. 107–117.

[9] Dimond, R., Mencer, O. & Luk, W. Application-specific customisation of multithreaded soft processors. Computers and Digital Techniques, IEE Proceedings 153(2006)3, pp. 173-180.

[10] Esko, O., Jääskeläinen, P., Huerta, P., de La Lama, C.S., Takala, J. & Martinez, J.I. Customized exposed datapath soft-core design flow with compiler support. Field Programmable Logic and Applications (FPL), Milano, Italy, August 31-September 2, 2010. pp. 217-222.

[11] Xtensa customizable processors [WWW]. Cadence Design Systems, Inc. [accessed on 11.5.2014]. Available at: http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[12] Kruijtzer, W., van der Wolf, P., de Kock, E., Stuyt, J., Ecker, W., Mayer, A., Hustin, S., Amerijckx, C., de Paoli, S. & Vaumorin, E. Industrial IP integration flows based on IP-XACT standards. Design, Automation and Test in Europe, Munich, Germany, March 10-14, 2008. pp. 32-37.

[13] El Mrabti, A., Petrot, F. & Bouchhima, A. Extending IP-XACT to support an MDE based approach for SoC design. Design, Automation and Test in Europe, Nice, France, April 20-24, 2009. pp. 586-589.

[14] Perry, T.P., Walke, R. & Benkrid, K. An extensible code generation framework for heterogeneous architectures based on IP-XACT. Southern Conference on Programmable Logic (SPL), Cordoba, Argentina, April 13-15, 2011. pp. 81-86.

[15] 1685-2009 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows [WWW]. IEEE. [accessed on 11.5.2014]. Available at: http://standards.ieee.org/findstds/standard/1685-2009.html.

[16] AMBA [WWW]. ARM, Ltd. [accessed on 11.5.2014]. Available at: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html.

[17] Matilainen, L., Kamppi, A., Määttä, J.-M., Salminen, E. & Hämäläinen, T.D. KACTUS2: IP-XACT/IEEE1685 compatible design environment for embedded Multiprocessor System-on-Chip products. Tampere 2011, Tampere University of Technology, Report 37, 2011.

[18] Matilainen, L., Lehtonen, L., Määttä, J-M., Salminen, E. & Hämäläinen, T.D. System-on-chip deployment with MCAPI abstraction and IP-XACT meta-data. 2012 International Conference on Embedded Computer Systems (SAMOS), Samos, Greece, July 16-19, 2012. pp. 209-216.

[19] Kactus2 [WWW]. Tampere University of Technology. [accessed on 11.5.2014]. Available at: http://funbase.cs.tut.fi/

[20] Fisher, J. Very long instruction word architectures and the ELI-512. Annual International Symposium on Computer Architecture, New York, NY, USA, 1983. pp. 140-150.

[21] TTA-based Co-design Environment v1.9 User Manual [Online document]. Tampere University of Technology, Department of Computer Systems. Published on 25.10.2006, last updated on 24.1.2014 [accessed on 11.5.2014]. Available at: http://tce.cs.tut.fi/user_manual/TCE.pdf.

[22] Wolf, W., Jerraya, A.A. & Martin, G. Multiprocessor System-on-Chip (MPSoC) Technology. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(2008)10, pp. 1701-1713.

[23] Multicore Communications API [WWW]. The Multicore Association. [accessed on 11.5.2014]. Available at: http://www.multicore-association.org/workgroup/mcapi.php.

[24] Kamppi, A., Matilainen, L., Määttä, J.-M., Salminen, E. & Hämäläinen, T.D. Extending IP-XACT to embedded system HW/SW integration. 2013 International Symposium on System-on-Chip, Tampere, Finland, October 23-24, 2013. Tampere 2013, Juvenes Print. 8 pages.

[25] Kamppi, A., Määttä, J.-M., Matilainen, L., Salminen, E. & Hämäläinen, T.D. Kactus2: Extended IP-XACT metadata based embedded system design environment. Embedded Systems Week / MeCoES: Metamodeling and Code Generation for Embedded Systems workshop, Tampere, Finland, October 7, 2012. pp. 17-22.

[26] SOPC Builder Support [WWW]. Altera Corp. [accessed on 11.5.2014]. Available at: http://www.altera.com/support/software/system/sopc/sof-sopc_builder.html

[27] Ramabadran, T.V. & Gaitonde, S.S. A tutorial on CRC computations. Micro, IEEE 8(1988)4, pp. 62-75.

[28] Quartus II Handbook Version 9.1 Volume 5: Embedded Peripherals [Online document]. 2009 [accessed on 12.5.2014]. Available at: http://www.altera.com/literature/hb/qts/archives/quartusii_handbook_9.1.2.pdf

[29] Hennessy, J.L. & Patterson, D.A. Computer Architecture: A Quantitative Approach. 4Th edition. San Francisco, CA, USA 2006, Morgan Kaufmann. 704 pages.