

### VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY DEPARTMENT OF COMPUTER SYSTEMS

## PROSTŘEDÍ PRO REKONFIGUROVATELNÉ SYSTÉMY NA ČIPECH ALTERA

FRAMEWORK FOR RECONFIGURABLE SYSTEMS ON THE ALTERA CHIPS

DIPLOMOVÁ PRÁCE MASTER'S THESIS

AUTOR PRÁCE AUTHOR

VEDOUCÍ PRÁCE SUPERVISOR Bc. BRUNO KREMEL

Ing. PAVOL KORČEK

BRNO 2015

### Abstrakt

Práce posuzuje dostupné prostředí pro vývoj rekonfigurovatelných systémů na čipech Altera. Tyto prostředí jsou následně porovnány s řešeními dostupnými pro platformu Xilinx. Prostředí RSoC Framework je pak představeno jakožto výhodná alternativa pro řešení vyvinuté výrobcemi. Prostředí je momentálně k dispozici na platformě Xilinx Zynq. Dále práce hodnotí klíčové rozdíly mezi platformou Xilinx Zynq a platformou Altera Cyclone V SoC a navrhuje způsob řešení portace uvedeného prostředí na platformu Altera. Následně se diskutuje návrh a implementace portu na platformu Altera Cyclone V SoC. Nakonec práce vyhodnocuje výkonnost portovaného systému na nové platformně.

### Abstract

This work reviews the development frameworks available for the *Altera* System-On-Chip solutions. These solutions are then compared to solutions available on the *Xilinx* platform. The *RSoC Framework* is then presented as an advantageous alternative for the vendor's solutions. This framework is currently available for the *Xilinx Zynq* platform. Furthermore the work assess the key differences between *Xilinx Zynq* and *Altera Cyclone V SoC* platforms and proposes the solution to port the framework to *Altera platform*. The design and implementation of then *RSoC Framework* port to *Altera Cyclone V SoC* is then discussed. Finally the work evaluates the performance of the ported system on the new platform.

### Klíčová slova

SoC, FPGA, Altera, Cyclone V SoC, Zynq, RSoC

#### **Keywords**

SoC, FPGA, Altera, Cyclone V SoC, Zynq, RSoC

### Citation

Bruno Kremel: Framework for Reconfigurable Systems on the Altera Chips, diplomová práce, Brno, FIT VUT v ${\rm Brn}\check{\rm e},\,2015$ 

# Framework for Reconfigurable Systems on the Altera Chips

### Declaration

I declare that this project is my own work and that I have properly acknowledged the use of work and information from other sources.

Bruno Kremel May 27, 2015

### Acknowledgement

I would like to thank Ing. Pavol Korček for being a supportive supervisor and Ing. Jan Viktorin for being helpful with technical details of the port of the framework and also for reviewing my work.

© Bruno Kremel, 2015.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

## Contents

1 Introduction				
2	Em	bedded systems, System-on-Chip and reconfiguration	4	
	2.1	Embedded systems	4	
	2.2	System-on-Chip and Reconfigurable System-on-Chip	4	
	2.3	Available frameworks for Reconfigurable System-On-Chip	6	
		2.3.1 Xilinx SDx Development Environments	6	
		2.3.2 Altera SDK for OpenCL	8	
		2.3.3 BSoC Framework	9	
	24	Overview of the BSoC Framework	10	
	2.1	2.4.1 Architecture of the BSoC Bridge	11	
3	Cha	aracteristics of Altera and Xilinx RSoC systems	13	
	3.1	Interfaces	13	
		3.1.1 AMBA AXI	13	
		3.1.2 Altera Avalon	16	
	3.2	Hard processor	19	
	3.3	DMA controller	21	
		3.3.1 Differences between Altera S&G DMA and Xilinx AXI DMA controller	22	
	3.4	Design tools	27	
		3.4.1 Synthesis and implementation tools	28	
		3.4.2 IP integration tools	28	
	3.5	Drivers	29	
		3.5.1 Standard Linux driver	29	
		3.5.2 Zerocopy RX Linux driver	29	
		3.5.3 Multiplatform driver	30	
4	Des	ign and implementation	32	
	4.1	Proposed solution	32	
	4.2	Components for integration of the Altera S&G DMA to the RSoC Bridge	33	
		4.2.1 RSoC DMA wrapper for the Altera S&G DMA	33	
		4.2.2 AXI — Avalon adaptors	35	
	43	Integration with Altera tools	42	
	1.0	4.3.1 Altera Quartus synthesis support	42	
		4.3.2 Altera Osys system integration tool support	<u> </u>	
		4.3.3 IP core distribution	11	
	1 1		44	
	4.4	4.4.1 Deceminton processing differences between Altere and Vilian drivers	40	
		4.4.1 Descriptor processing differences between Aftera and Allinx drivers .	40	

<b>5</b>	Test	ting and measurements	<b>48</b>			
	5.1	Hardware platform	48			
	5.2	Simulation and unit tests	50			
	5.3	Performance testing and measurements	50			
		5.3.1 Throughput measurements	50			
		5.3.2 Resource utilisation	53			
6	Conclusion					
Α	Contents of the included CD 5					
В	Hov Dev	v to build and use the RSoC Framework on the Altera Cyclone V SoC relopment Board	59			
	B.1 B.2 B.3	Compiling the Quartus design for the Altera Cyclone V SoC Development Board	59 60 61			

## Chapter 1

## Introduction

Devices based on *System-on-Chip* makes notable share of the market today. The *System-on-Chip* evolved from embedded systems of former discrete microprocessors and discrete logic (e.g. 74 series). Continued through the integrated microcontrollers to full fledged computing systems used in today's cell phones. Applications vary from systems that could be considered embedded: the TV Set top boxes, digital cameras, intelligent IP cameras to more or less universal computing systems like smartphones, tablets, smartwatches, smart TVs or car dash computers.

For all applications, embedded system basically consists of an *application specific logic* designed for particular task and some *microprocessor*. Today the *application specific logic* is often integrated on the same die as the microprocessor, these systems are called *System-on-Chip* (SoC), mostly based on the *ARM* architecture and implemented as an *ASIC*.

Leading programmable logic manufacturers understood the need for a *reconfigurable* System-on-Chip and embraced the concept of the System-on-Chip for their programmable logic products creating an ARM and FPGA based System-on-Chip solution.

To design an application for these systems one must design an FPGA firmware, drivers for an operating system (OS) often based on Linux or some Real Time OS and an application software. This complexity calls for some collection of a tools and IPs that will allow designers to focus only on an application specific design. One of these collections is the RSoC Framework currently available for the Xilinx SoCs.

To provide designers with a choice of devices, *RSoC Framework* needs to be available on more than a single platform. Choice has been made to port the framework to *Altera SoCs*. The reason is that *Altera* and *Xilinx* competes for the position of the largest programmable logic manufacturer with an alternating leading position.

This work explains reasons behind utilisation of the reconfigurable System-on-Chip in embedded systems. Introduces the RSoC Framework for these systems and explains main advantages of this framework in development for these systems. Further text covers the key differences between Xilinx and Altera reconfigurable System-on-Chip platforms, their impact on the framework and propose a solution to port the current implementation over to the Altera Cyclone V SoC platform. Finally the work covers how this solution was implemented and summarises the results of tests and measurements of resulting system.

### Chapter 2

## Embedded systems, System-on-Chip and reconfiguration

Following chapter covers what is an embedded system, explains what makes a Systemon-Chip suitable for some applications of embedded systems and deals with the difference between a System-on-Chip and reconfigurable System-on-Chip. A brief comparison of frameworks for reconfigurable System-on-Chips will be presented. Finally the RSoC Framework for reconfigurable System-on-Chips is introduced.

#### 2.1 Embedded systems

Embedded system is considered a computing system embedded in larger physical system like washing machine, microwave, TV set to box etc, where user is not aware that this system actually contains a computing system and embedded system itself is designed for single particular task. Essentially embedded system consists of a microprocessor and some application specific logic.

However especially systems like smartphones, tablets, smart TVs etc. makes the term embedded hard to *distinguish*. Take for example today's smartphones, they are a *universal computing system* rather than an *embedded system* and application specific logic is not used for single particular task rather than accelerating set of common tasks. And in fact you can find the *same SoC device* in clearly an embedded platform for example an intelligent IP camera and in a platform that couldn't be considered embedded anymore such as a smartphone.

#### 2.2 System-on-Chip and Reconfigurable System-on-Chip

**System-on-Chip** Today's market (especially smartphone and tablet market) called for more *integration* to reduce *size*, *complexity* and *power consumption* of a given device that led to development of a *System-on-Chip*.

In these systems, the application specific logic is integrated on the same die as the *microprocessor*. It is implemented in an *ASICs* as it is oriented on accelerating a common set of tasks such as audio/video codecs, encryption etc. Also these systems shares common peripherals like Bluetooth, WiFi, GSM radio and others.

Therefore System-on-Chip is a perfect solution to get low power consumption and small size while maintaining high performance in certain applications. But these requirements are also common for embedded systems and therefore SoC are being gradually deployed in embedded systems as well.



Figure 2.1: Typical System-on-Chip

As shown on the Figure 2.1, these systems are logically divided to a *microprocessor*, on-chip *interconnect* and a set of *accelerators and peripherals* connected through this interconnect. The application specific logic is implemented by these accelerators/peripherals.

The idea behind a *System-on-Chip* is that a *microprocessor* is common and licensed (for example ARM Cortex cores). The device manufacturer will develop their own set of *accelerators and peripherals* creating a custom *SoC*. Example of such approach are Apple's Ax series or Samsung's Exynos series developed for smartphones and tablets of respective manufacturers.

**Reconfigurable System-on-Chip** The nature of some (especially embedded) applications creates the need for more *specialised* logic. The set of tasks for such system is not always common to a smartphone or a tablet applications and *common SoCs* are therefore *not an viable option*. Also only mayor electronics manufacturers are able to develop and roll out their own ASIC SoC. And even then an ASIC based SoC could be highly *uneconomical*, because some embedded system might not be manufactured in a volume that would cover the cost of an ASIC development.

To make an alternative for these applications, mayor programmable logic manufacturers embraced a System-on-Chip solution for their programmable logic. Creating an FPGA based Reconfigurable System-on-Chip where a microprocessor, some peripherals and a controllers are implemented as an ASIC and an application specific logic is implemented as a circuit in an FPGA as depicted on the Figure 2.2. Reconfigurable SoCs are also useful as a prototyping platform for development of an ASIC based SoC.



Figure 2.2: Typical Reconfigurable System-on-Chip

#### 2.3 Available frameworks for Reconfigurable System-On-Chip

All programmable logic manufacturers do their best to make development for their devices simple and straightforward by providing integration tools and *IPs*. However, as mentioned in the introduction, development for these *RSoCs* is *still challenging*. To make an *RSoC* application, one must develop a mean to make a reliable communication between the custom hardware *application specific logic* and the application *software* with some additional requirements such as high throughput and low-latency.

A complex firmware for an FPGA, software drivers for an operating system used (often Linux or some RTOS) must be developed to get a communication between an accelerator and an application software.

To let application designer focus *solely* on their specific application it is best to use a collection of tools and *IP*s that allow designer to only develop their *specific application logic* and *application software*.

Such frameworks exists for both platforms. The vendor provided frameworks are focused on high-level application development. Further sections will briefly introduce them. Then the *RSoC Framework* will be compared to these frameworks, the advantages and disadvantages of both approaches will be discussed as well.

#### 2.3.1 Xilinx SDx Development Environments

Xilinx provides three different development environment in their SDx family. These environments are currently in development and only early access is available to Xilinx partners. Therefore the following information being Xilinx advertising material [18] is preliminary in terms of real capabilities of these environments.

**SDSoC Environment** This environment is specifically targeted for the *Reconfigurable* System-On-Chip in this case the Zynq family. Xilinx promises [17] that SDSoC provides C/C++ full-system optimising compiler, delivers system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and libraries to speed programming.



Figure 2.3:



The development approach (illustrated on the Figure 2.3) is that the user writes an application in C/C++ and the SDSoC tools will *automatically* analyse bottlenecks with a profiler and then converts these parts of code into an FPGA design (utilising Vivado HLS). The communication system between the FPGA and the software is also generated automatically.





Figure 2.6: The development flow with Xilinx SDAccel [16]

**SDAccel Environment** This environment leverages OpenCL which brings the GPGPU like experience to FPGA accelerator card development. The target of this environment are accelerator cards in data-centres.

Xilinx advertises [16] that the SDAccel environment provides an architecturally optimising compiler supporting any combination of OpenCL, C, and C++ kernels. The development flow is show on the Figure 2.5.



SDNet - Software Defined Specification Environment for Networking

Figure 2.7:

Figure 2.8: The use of SDNet [19]

**SDNet Environment** The last environment is *Xilinx* approach to Software Defined Networking. As *Xilinx* material states [19] the *SDNet* stands for 'Softly' Defined Networks. This supports the SDN functionality, but also adds software programmable data plane hardware.

The use of SDNet is illustrated on the Figure 2.7. Xilinx claims: "SDNet enables the ability to use high-level specifications in conjunction with application optimised libraries and the associated design environment to automatically transform the specifications into an optimised hardware implementation on Xilinx devices." [19].

#### 2.3.2 Altera SDK for OpenCL

To simplify development for their devices Altera provides an OpenCL SDK, which targets their FPGAs. This environment is similar to the Xilinx SDAccel environment, but addresses both accelerator cards and the Reconfigurable System-On-Chip. This makes the Altera solution more unified, but the Altera solution is focused on solely on OpenCL as opposed to Xilinx solutions. The OpenCL is more known for being used in GPGPU applications. However OpenCL is an unified programming model for accelerating algorithms on heterogeneous systems.

In their SDK Altera provides following (as advertised here [2]):

- An emulator to step through the code on an x86 and ensure it is functionally correct.
- A detailed optimisation report to understand the load and store inner loop dependencies.
- A profiler that shows performance insight into the kernel to ensure proper memory coalescence and stall free hardware pipelines.
- An OpenCL compiler capable of performing over 300 optimisations on the kernel code and producing the entire FPGA image in one step.

The expected development flow is shown on the Figure 2.9. As we can see the developer can optimise the OpenCL application for the FPGA using emulator and target the FPGA after the application is optimised. This is important because compilation for the FPGA will take considerable time.



Figure 2.9:

Figure 2.10: The development flow with Altera OpenCL [2]

#### 2.3.3 RSoC Framework

Development with previously presented tools (the Altera SDK for OpenCL and Xilinx SDx Development Environment) is completely different from traditional RTL-based design approach for FPGA applications. Unlike the RSoC Framework, they are solely focused on acceleration of the computation.

While it has many advantages, such as getting programmers who developed for example GPGPU applications to seamlessly transition to developing applications for FPGAs. Development in software emulators and targeting the FPGA, after the application is optimised is also beneficial.

It has also disadvantages. The first is that RTL development can not be done with these tools. Another is that the high-level compiler either HLS or OpenCL can not always use FPGA resources effectively. And finally not all applications could be done with the SDx or OpenCL, take for example a high-speed data acquisition. This can not be easily written in OpenCL or HLS languages.

The RSoC framework has a different approach. It retains the low level RTL development, while letting user use HLS where it makes sense. With this approach, the development for *Reconfigurable System-On-Chip* is simplified for both RTL designers or HLS programmers.

Another advantage of the *RSoC Framework* is that, it is not tied to a single platform. An application written with *RSoC Framework* can be seamlessly used on different platforms.

#### 2.4 Overview of the RSoC Framework



Figure 2.11: Basic design of RSoC Bridge

The framework as depicted on the Figure 2.11 gives a stream based access to accelerators. Developer only needs to know a user-space interface and an accelerator interface of the bridge.

As the original work states [21] the main goal of the RSoC Framework is to encapsulate the details of the target platform and provide user with a simple stream based communication between the software and the application logic. A software application uses the standard read/write system calls provided by an operating system. An application logic uses simplex streams to exchange frames with software. The RSoC Framework uses the RSoC Bridge layer which ensures that the data is transferred between an application logic and the software.

Following text will make a brief overview of the *RSoC Framework* only with details that are necessary for further sections of this work. The *RSoC Framework* defines following components:

**RSoC Accelerator** The framework defines an accelerator as an interface.

As shown on the Figure 2.12 RSoC Accelerator utilises:

- A streaming interface (AXI Stream).
- A 32bit information vector (to get information about accelerator).
- An optional configuration interface (AXI4-Lite or AXI4-Full).



Figure 2.12: RSoC accelerator interface

• An optional 4bit event vector (interrupt requests from accelerator).

The user only needs to adapt his application specific unit to use this RSoC Accelerator interface, user's unit does not necessary have to accelerate anything. User's unit can have additional interfaces and can be used as an *peripheral adaptor* or be an *interface adaptor* to a whole system that could utilise a stream based transfer between a programmable logic and a microprocessor. In fact this system could use multiple RSoC Accelerators if needed.

**RSoC Bridge** The RSoC Bridge is a layer between a hard processor and the RSoC accelerators. The RSoC Bridge implements an interconnect between the accelerators and a hard processor and provides data transfer engine to transfer data between *software* and RSoC accelerators, it also maps the events from the RSoC Accelerators to the hard processor.

**RSoC Driver** The RSoC Driver provides an interface to the user-space applications. Each RSoC Accelerator is represented by a matching character device. User transfer frames to and from the RSoC Accelerator by standard OS calls.

#### 2.4.1 Architecture of the RSoC Bridge

The RSoC Bridge is the essential part of the RSoC Framework and it is necessary to understand it's architecture. On the Figure 2.13 we can see that every RSoC Accelerator has assigned a transfer engine. This transfer engine is currently implemented by an DMAcontroller. Bridge also have to merge all links from transfer engines and interconnect them to hard processor interface, links from hard processor have to be merged and split to each accelerator as well. This interconnect system is currently based on the AXI4 bus, as ARMuses this bus on their SoC microprocessor solutions.



Figure 2.13: The basic RSoC Bridge architecture

**DMA interface** The RSoC Framework provides a common interface for the DMA controllers. Every DMA controller that is going to be used with the RSoC Bridge needs to conform to this interface. As seen on the Figure 2.14, this interface provides a master



Figure 2.14: Interface for DMA controllers

memory access link M\_MEM, a master descriptor access link M\_SG and a slave control link S\_AXI. These links are *AXI4-Full* for master links and *AXI4-Lite* for slave link. Interface also provides master and slave *AXI Stream* links for streams.

### Chapter 3

## Characteristics of Altera and Xilinx RSoC systems

For successful port to a new platform, it is necessary to assess differences between these platforms. It is important to compare *hard processor*, *peripherals* and hard processor's *interconnect* to an *FPGA* and also design tools. This chapter therefore compares the *Xilinx* and the target platform from different aspects. The comparison will be more focused on the target platform — *Altera*.

Selected parameters shown in the Table 3.1 from Altera white-paper[7] presents a brief comparison of the Xilinx and Altera SoCs. Where LE are Logical Elements and LC

Parameter	Altera	Xilinx
Processor	ARM Cortex-A9	ARM Cortex-A9
FPGA fabric	Cyclone V, Arria V	Artix-7, Kintex-7
FPGA Logic Density Range	$25~\mathrm{K}$ to $462~\mathrm{K}~\mathrm{LE}$	$28~\mathrm{K}$ to $444~\mathrm{K}~\mathrm{LC}$

Table 3.1: Parameters of the Altera and Xilinx SoCs

are Logical Cells. According to the Altera comparison [10], the ratio of 1.125:1 should be used in favour of Altera LEs when comparing device densities. However this comparison is outdated and made by Altera, therefore it's credibility is questionable. But still LCs and LEs count should be fairly comparable since they measure similar elements – equivalent of four-input LUT and a flip-flop [1] [15] and other logic like carry logic and registers.

#### 3.1 Interfaces

Both platforms are based on ARM and share a common interface – AXI. However, since *Altera* is using *Avalon* on their *Nios II* soft microprocessor platform, many *IP* cores for the *Altera* platform remain implemented only with *Avalon* interfaces. Further text covers characteristics of each interface.

#### 3.1.1 AMBA AXI

The AMBA AXI is a set of point-to-point protocols:

- AXI3, AXI4 and AXI4-Lite an address and burst based read/write protocol
- AXI4-Stream an stream-based unidirectional (simplex) protocol

#### AXI

The AXI protocol [5] itself defines following independent transaction channels:

- read address
- read data
- $\bullet\,$  write address
- write data

An address channels also carries control information that describes the nature of the data to be transferred. The data is transferred using either:

- A write channel to transfer from the master to the slave. The slave uses the write response channel to acknowledge completion to the master.
- A read channel to transfer from the slave to the master.

The AXI protocol permits that the address information is issued ahead of the actual transaction, supports multiple outstanding transactions and supports out-of-order completion of the transfers. The Figure 3.1 shows how the write transaction uses the address channel to



Figure 3.1: Basic architecture of the AXI write transaction [5]

initiate the transaction and the write channel to transfer the data and the response channel to acknowledge that the data has been written. The Figure 3.2 shows how the read



Figure 3.2: Basic architecture of the AXI read transaction [5]

transaction uses the address channel to initiate the transaction and the read data channel to transfer the data. Note that response channel is not necessary, as the data itself acknowledges the transaction completion.

Signal Source Semantics AxADDR Master Read/write address. It gives the address of the first transfer in a burst. MasterBurst length, exact number of transfers in a burst is  $A \times L \in N + 1$ . AxLEN MasterBurst size. It indicates size of each transfer in the burst. AxSIZE AxBURST Master Burst type. AxVALID Master Indicates that channel is signalling a valid address and control information. Slave Indicates that slave is ready to accept an address and control signals. AxREADY MasterWrite data. WDATA WSTRB Master Indicates which byte lanes hold valid data. There is one bit for each byte of data bus. WLAST MasterIndicates the last transfer in a write burst. WVALID MasterIndicates that valid write data and strobes are available. Slave Indicates that the slave can accept the write data. WREADY Slave BRESP Indicates the status of the write transaction. BVALID Slave Indicates that the channel is signalling a valid write response Indicates that the master can accept a write response. Master BREADY Slave Read data. RDATA Slave Indicates the status of the read transaction. RRESP RLAST Slave Indicates the last transfer in a read burst. Slave Indicates that the slave is signalling the required read data. RVALID RREADY MasterIndicates that the master can accept a read data and response information.

The Table 3.2 sums up the important signals and their semantics for AXI4 [5]. Additional

Table 3.2: Overview of the AXI4 signals

signals not included in the overview are: clock (ACLK), reset (ARESETn), IDs (xID, AXID), user (AXUSER, XUSER), memory type (AXCACHE), protection (AXPROT) and lock (AXLOCK). Their semantic can be studied in the ARM documentation [5].

#### AXI4-Lite

AXI4-Lite is a subset of AXI4 protocol. Key features:

- All transactions are of burst length 1 (equivalent to AxLEN being always zero, AxBURST being always zero, xLAST being always one).
- All data accesses use the full width of the data bus (AxSIZE is defined as bus width).
- Exclusive accesses are not supported (equivalent to AxLOCK value of zero).
- All accesses are non-modifiable, non-bufferable (equivalent to an AxCACHE value of 0b0000).

The signals AXLEN, AXSIZE, AXBURST, AXLOCK, AXCACHE and XLAST are not supported by the *AXI4-Lite* interface.

#### AXI4-Stream

The AXI Stream is a simplex streaming protocol with a back-pressure and no acknowledge. The following Table 3.3 sums the signals and their semantics. As specification [4] in the

Signal	Source	Semantics
TVALID	Master	(1) indicates that the master is driving valid transfer
TREADY	Slave	(1) indicates that slave can accept a transfer
TDATA[(8n-1):0]	Master	Is a primary data payload. Width is an integer number of
		bytes.
TSTRB[n-1:0]	Master	Indicates whether associated byte is a data $(1)$ or a position
		byte (0).
TKEEP[n-1:0]	Master	Indicates whether associated byte is a part of the data
		stream.
TLAST	Master	Indicates boundary of a packet.
TID[i-1:0]	Master	Is data stream identifier.
TDEST[d-1:0]	Master	Provides routing information.
TUSER[u-1:0]	Master	Is an user defined sideband information.

Table 3.3: Overview of the AXI4-Stream signals [4]

Section 3.1 Default value signalling states only ACLK, ARESETN and TVALID signal is required, other signals can be assumed having a default value when not used. Therefore as per standard this interface may not carry any payload.

#### 3.1.2 Altera Avalon

The *Avalon* is a family of interfaces. *Altera* specifies 7 interface roles of which two are of interest in this work:

- Avalon Memory Mapped Interface (*Avalon-MM*) an address-based read/write capable of burst and pipelined transfers.
- Avalon Streaming Interface (Avalon-ST) an stream based interface that supports unidirectional (simplex) flow of data.

These two interfaces have rough counterparts in the AXI family of protocols.

The others: Avalon Conduit Interface, Avalon Tri-State Conduit Interface, Avalon Interrupt Interface, Avalon Clock Interface, Avalon Reset Interface are of no interest, since they have no counterparts in AXI specification and are logical specification of interface for Altera Qsys integration tool, rather than a proper protocol definition.

#### Avalon-MM

Avalon-MM range from simple to complex interfaces. The basic principle is that the master initiates a transaction by asserting either of read or write signals and slave can stall the transaction by asserting the waitrequest signal. When slave is ready the waitrequest signal is de-asserted.

The Table 3.4 shows *Avalon-MM* signals and their semantics [8]: Further text covers typical *Avalon-MM* interface types.

Signal	Source	Semantics
ADDRESS	Master	Represents a byte address, must be aligned to the data width.
BYTEENABLE	Master	Enables specific byte lane. Bit $n$ indicates whether byte $n$ is being
		transferred.
WRITE	Master	Asserted to indicate a write transfer.
READ	Master	Asserted to indicate a read transfer.
WRITEDATA	Master	Data payload for write transfer.
READDATA	Slave	Data payload for read transfer.
READDATAVALID	Slave	Indicates that the readdata signal contains valid data. Must be as-
		serted for one cycle for each read access received.
WAITREQUEST	Slave	Asserted by the slave when it is unable to respond to a read/write re-
		quest. Forces the master to wait until the slave is ready.
BURSTCOUNT	Master	Indicate the number of transfers in each burst, minimal value is 1. Value
		must be a power of 2.

Table 3.4: Overview of the Avalon-MM signals

Non-pipelined, non-bursting Avalon-MM interface with slave controlled waitrequest It is a typical *Avalon-MM* interface that supports read and write transfers with a slave controlled waitrequest signal. The Figure 3.3 shows a typical transaction, where a slave



Figure 3.3: Read and write transfer with waitrequest [8]

can stall transfer by holding the waitrequest signal. The signals READDATAVALID and BURSTCOUNT are not used in this type of interface. This interface approximately corresponds to AXI4 Lite interface.

Non-pipelined, non-bursting Avalon-MM interface with Fixed Wait-States It is a modification of the previous interface where Wait-States are generated internally by both, the master and slave instead of using the waitrequest signal. The length of these Wait-States are defined by writeWaitTime and readWaitTime interface properties. This value mimics behaviour as if the waitrequest signal was asserted for n cycles. This interface is illustrated on the Figure 3.4 on which values of writeWaitTime = 2 and readWaitTime = 1 are used.



Figure 3.4: Read and write transfer with Fixed Wait-States at Slave interface [8]

**Pipelined, bursting Avalon-MM interface** The Figure 3.5 shows a typical bursting read transaction and the Figure 3.6 shows a typical bursting write transaction. Note that this type of interface supports bursts and multiple outstanding transactions (pipelined), although only for read transactions, write transactions only support bursting transactions. This type of interface capability-wise corresponds to the *AXI* interface. The out-of-order completion is not supported.



Figure 3.5: Bursting read transaction [8]



Figure 3.6: Bursting write transaction [8]

**Avalon-ST** The *Avalon-ST* interface is a streaming interface with following features:

• Unidirectional (simplex) protocol.

- Multiple channel support.
- Sideband signalling of channel, error and start and end of packet.

Signal	Source	Semantics
CHANNEL	Source (Master)	The channel number for data being transferred.
DATA	Source (Master)	The data payload.
ERROR	Source (Master)	A bit mask used to mark errors affecting the data
		being transferred in the current cycle.
VALID	Source (Master)	Qualifies all other source's signals.
EMPTY	Source (Master)	Indicates the number of symbols that are empty dur-
		ing cycles that contains the end of packet. Always
		last symbols in DATA.
ENDOFPACKET	Source (Master)	Marks the end of packet.
STARTOFPACKET	Source (Master)	Marks the start of packet.
READY	Sink (Slave)	Indicates that sink can accept the data.

Table 3.5: Overview of the Avalon-ST signals

The signal CHANNEL is optional, also when no packet transfer is necessary, the signals ENDOFPACKET, STARTOFPACKET and EMPTY could be omitted. As well as READY when no backpressure is needed or VALID when Source implicitly provide valid data on every cycle.

#### 3.2 Hard processor



Figure 3.7: Basic architecture common to Xilinx and Altera FPGA SoCs

A hard processor is a collection of ASICs that makes the CPU, peripherals and the interconnect to the FPGA. The design in the FPGA is interfacing with this system, therefore it is important to know it's basic architecture. A hard processor is called Hard Processor System by Altera and Processing System by Xilinx.

Both vendors based their SoCs on the ARM Cortex-A9 microprocessor system, therefore they share a common basic architecture shown on the Figure 3.7 with differences generally in the terminology, number of ports, memory sizes etc.

Following tables based on *Altera* white-paper [7] show a brief comparison of *Altera* [11] and *Xilinx* [20] terminology and port configuration available on respective vendor's *SoCs*.

Table 3.6 shows a comparison of the interconnect between the CPU and the FPGA.

Parameter	Altera	Xilinx
High-Bandwidth Processor/FPGA Interconnect	$1 \ge 32/64/128$ bit AXI	2x 32 bit AXI
	(CPU to FPGA)	(CPU to FPGA)
	$1 \ge 32/64/128$ bit AXI	2x $32$ bit AXI
	(FPGA to CPU)	(FPGA to CPU)
Low-Latency Processor/FPGA Interconnect	1x 32 bit AXI	not available
	(CPU to FPGA)	(shared with high-
		bandwidth)
Processor/FPGA interconnect data width (theo-	32/64/128 (10.8 GB/s)	32 (4.8 GB/s)
retical max. bandwidth)		

Table 3.6: Processor-to-FPGA system interconnect characteristics

Following Table 3.7 compares the interconnect of the FPGA and the memory controller.

Parameter	Altera	Xilinx
FPGA-to-DDR Memory interconnect path	256 bit, AXI/Avalon-MM	4x 64 bit AXI
	interface	(FPGA to DRAM
	(FPGA to DRAM)	and on-chip
		RAM)
Individual port size options	8/16/32/64/256 bit	32/64 bit
Maximum FPGA to interconnect ports	6 command/response ports	4 x64 read ports
	4 read ports	4 x64 write ports
	4 write ports	
Maximum interconnect to processor DDR	6 command/response ports	2 x64 read port
Hard Memory Controller ports (connection	4 read ports	$2 \times 64$ write port
type)	4 write ports	(multiplexed)
	(direct)	

Table 3.7: Memory-to-FPGA system interconnect characteristics

Finally the Table 3.8 shows the different terminology used by Altera and Xilinx.

Port	Altera	Xilinx
High-Bandwidth Processor/FPGA Interconnect	HPS-to-FPGA (H2F)	General Purpose
(CPU to FPGA)	FPGA-to-HPS (F2H)	$(AXI_GP)$
(FPGA to CPU)		
Low-Latency Processor/FPGA Interconnect	HPS-to-FPGA	not available
(CPU to FPGA)	${ m Lightweight}$	
	(H2F LW)	
ACP	FPGA-to-HPS	ACP (AXI_ACP)
FPGA-to-DDR Memory interconnect port	FPGA-to-HPS	High Performance
	SDRAM	$(AXI_HP)$
	(F2H SDRAM)	

Table 3.8: Overview of interface the terminology differences of Altera and Xilinx

**Central (Xilinx) or L3 (Altera) interconnect** Both vendors utilises the *ARM AMBA NIC-301* in their interconnect. The design of this interconnect is critical for performance. It is always a compromise of cost, latency and throughput. As stated in [7] *Altera* tries to minimise hierarchy of this interconnect to minimise latency.

Also the *Altera SoC* platform has dedicated low-latency master bus which can be beneficial in some applications with high bandwidth between the CPU and the FPGA fabric.

Furthermore Xilinx uses a QoS-based arbitration and Altera use a priority based (per master) with least recently used (LRU) algorithm for masters with equal priority. Both types of arbitration can be useful in different set of task, although arguably for embedded applications (especially real time systems) the priority based method used by Altera can be beneficial, because it is more deterministic than a QoS based system. However in most applications the latency of the interconnect will probably be negligible compared to other sources of latency.

#### **3.3 DMA controller**

To transfer the data effectively a DMA controller is often used in various applications. The DMA controller is also a critical part of the RSoC Bridge. Traditionally it is a hardware device which performs the data transfers as a bus master (either from peripheral to memory, from memory to peripheral or from memory to memory). The CPU only sets the parameters of the transfer.

**Traditional approach** Traditional DMA controllers are controlled directly by the CPU transaction-by-transaction. That means that CPU must set parameters of each transaction, wait for completion and read response. This type of controller can only queue one transaction at a time. The CPU load is substantial for small transfers, degrading overall system performance. The advantage is the simplicity of such controller and its programming model.

Scatter-Gather DMA controller A more advanced type of a DMA controller is a Scatter-Gather controller. This type of DMA controller does read a list of descriptors autonomously. Each descriptor holds the information about corresponding transaction. This leaves the CPU to just update the contents of the descriptor list. This makes the CPU do less processing and bus communication for each transfer and in turn increasing

the overall system performance and performance of the controller. Such controller is also suitable for the most of the RSoC Bridge applications. The only disadvantage is the complexity of such DMA engine (the hardware itself and the software controller).

#### 3.3.1 Differences between Altera S&G DMA and Xilinx AXI DMA controller

The current implementation of RSoC Bridge on Xilinx uses a proprietary netlist based Xilinx AXI DMA controller implemented in an FPGA which can not be used on the target platform due to both licensing and incompatibility with Altera devices and design tools. Therefore we will cover the Altera alternative and compare it to the Xilinx counterpart.

We can find similar interface roles (illustrated on Figures 3.8 and 3.9) between Xilinx and Altera Scatter-Gather DMA — both controllers have Scatter-Gather master memory access interface that is used by the DMA controller to access descriptors.

Second interface is either master memory access read or master memory access write (or both in case of *Xilinx AXI DMA*). There are also two (or one) streaming interfaces (master and slave).







Figure 3.9: Altera DMA controller interfaces

The main architectural difference between Xilinx and Altera is that Altera have separate units for upstream (*Stream-to-Memory*) and downstream (*Memory-to-Stream*) *DMA*, but Xilinx has one monolithic unit for both upstream (called S2MM by Xilinx) and downstream (*MM2S*). The *Altera DMA* can be also configured for *Memory-to-Memory* operation, but this is not interesting for the *RSoC Bridge*.

Xilinx S & G DMA uses AXI interfaces, which are used in the RSoC Framework internally. On a contrary, Altera S & G DMA core uses the Avalon-MM interface. The main differences between these interfaces were described in the Section 3.1.

Apart from the interfaces and architecture the key to understand the differences of the operation of given Scatter-Gather DMA controller is to understand its descriptor processing. The descriptors holds the information about what should be transferred where and how, therefore they are (apart of control registers) of most concern to the software driver. Following paragraphs will therefore apart from architectural differences also compare the descriptor processing of the each particular DMA controller.

#### Xilinx AXI DMA controller



Figure 3.10: AXI DMA Block Diagram [9]

The overview of the Xilinx AXI DMA architecture is shown on the Figure 3.10. As stated in the AXI DMA documentation [9] the Scatter-Gather capability of this controller is optional (but enabled in the RSoC Bridge). Primary DMA data movement is through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory (S2MM) Slave to AXI4 Write Master. The MM2S and (S2MM) channels operate independently. The Xilinx AXI DMA provides automatic burst mapping, as well as providing the ability to queue multiple transfer requests.

From the architectural point of view the AXI DMA controller consists of Data movers and associated control and status logic. This logic is controlled by a single scatter-gather controller which fetches the descriptors and issues commands to Data movers. When a data mover completes, the transfer the scatter-gather controller updates associated descriptor in the memory.

**Xilinx DMA controller descriptor processing** As shown on the Figure 3.11 the basic operation of *Xilinx DMA* is as follows [9].



Figure 3.11: Xilinx DMA controller descriptor processing

Software must first set up a descriptor chain (that is often a descriptor ring) and set up the *tail descriptor pointer* and *current descriptor pointer* (which are registers in the *DMA* controller).

The DMA controller then processes every descriptor from the point of *current descriptor* pointer up until the point it reaches the *tail descriptor pointer*, then DMA controller is idle until software updates the *tail descriptor pointer*.

Software then can process every buffer associated with completed descriptors and reallocate descriptors for the DMA controller (and update the *tail descriptor pointer*).

#### Available DMA controllers on the Altera platform

Altera provides two different S&G DMA controllers [14]. Older one the Altera S&G DMA and the newer one Alters Modular S&G DMA available since Quartus 14.0 (was not available while designing the RSoC Bridge port to Atera and therefore it was not used in favour of the older one). There is also a non-Scatter-Gather controller available, but it is not interesting for the RSoC Bridge port.

Altera modular Scatter-Gather DMA controller Altera presents this DMA controller as a fix to architectural issues in original Altera S&G DMA. The main architectural difference is its modular nature.

It consists of a dispatcher block with optional read and write master blocks. The dispatcher block fetches the descriptors and issues instruction to the read master and write master blocks.

While a single dispatcher can be used for both Read and Write Master (like in case of *Xilinx* controller), *Altera* recommends using separate dispatchers for upstream and downstream except when using Memory-Mapped to Memory-Mapped configuration.

The interface follows previous overview, but there is also an additional interface for response channel (optional).



Avalon-ST to Memory-Mapped

Figure 3.12: Altera Modular Scatter-Gather DMA Block Diagram [14]

With these three units *Altera* recommends three configurations of the *Modular* S & G *DMA* (as illustrated on Figure 3.12):

- Avalon-ST to Memory-Mapped.
- Memory-Mapped to Avalon-ST.
- Memory-Mapped to Memory-Mapped.

Where corresponding blocks are used together to create desired functionality.

The disadvantage of using a single dispatcher is that, unlike the Xilinx AXI DMA controller this controller has a single descriptor queue for both read and write channel. The

programming model could become unnecessary complex and therefore it might be more reasonable to use two dispatchers.

Altera Scatter-Gather DMA controller Similar to the modular controller, the older Altera S&G DMA controller consist of three functional blocks. The Descriptor Processor, the DMA Read Block and the DMA Write Block.

The descriptor processor reads descriptors via Avalon Memory-Mapped read master port and pushes commands into the command FIFOs of the DMA read and write blocks. After each *command* is processed by the DMA read or write block the associated descriptor is updated with the information about completed transfer.

The DMA read block reads commands from input command FIFO and issues read burst (if bursts are enabled) on its master Avalon-MM interface and pushes the read data to its source Avalon-ST port.

The DMA write block similarly reads commands from input command FIFO and pulls the data from its sink Avalon-ST port and in turn issues write bursts (if bursts are enabled) on its master Avalon-MM interface. The controller can be configured with either DMA read



Figure 3.13: Altera Scatter-Gather DMA Block Diagram [14]

block, DMA write block (or both in case of memory to memory configuration) as illustrated on the Figure 3.13.

Although these modules are very similar to the modular controller, the configuration of these modules is fixed to following three use cases available (distributed as monolithic units):

- Stream to memory (which is used as *upstream* DMA).
- Memory to stream (which is used as *downstream* DMA).
- Memory to memory (not used for the *RSoC Bridge*).

Altera S&G DMA controller descriptor processing The Altera DMA controller descriptor processing is shown on the the Figure 3.14. The key difference between the Xilinx

DMA controller and Altera is that the Altera DMA controller does not use a tail descriptor pointer. The operation of the Altera DMA controller is as follows [14].

The software must first set up a descriptor chain (it must specifically flag all used descriptors as *owned by hardware*) and set up the *next descriptor pointer* register and start the DMA controller.

![](_page_29_Figure_2.jpeg)

Figure 3.14: Altera DMA controller descriptor processing

The DMA controller then processes every descriptor from the *next descriptor pointer* up until the first descriptor that has *owned by hardware* flag set to 0. Then the DMA controller stops.

Note that if it stops, it must be restarted to start processing again. Simply setting *owned by hardware* flag on the descriptor on which the controller stopped to 1 will not restart descriptor processing.

The software then can process every buffer associated with processed descriptors that are no longer set as *owned by hardware* and can reset the flag *owned by hardware* to reallocate descriptors for the *DMA* controller (and possibly reset the *DMA* controller if it has already stopped).

#### 3.4 Design tools

The framework is designed to be platform *independent*. However there are also inevitably platform dependent parts: already mentioned DMA controller, top level design and IP integration user interface.

Also the user of the *RSoC Bridge* will always have to deal with vendor's design tools and therefore there must be a platform dependent layer. The following text will briefly introduce the *Altera* design tools and compares them to *Xilinx* tools.

#### 3.4.1 Synthesis and implementation tools

Altera provides the Quartus II software, while Xilinx provides the newer Vivado Design Suite and older Xilinx ISE. The Altera application note [6] shows a brief comparison of the

![](_page_30_Figure_2.jpeg)

Figure 3.15: Basic design flow

design flow shown on the Figure 3.15.

An interesting thing to note is that synthesis and mapping is a process which is done by quartus\_map command (called simply Analysis & Synthesis in the Quartus GUI).

#### 3.4.2 IP integration tools

Both vendors provide a system integration tool, that is graphical interface tool in which user can configure IPs and put together a system from vendor's and third-party IPs. To provide user a consistent and familiar environment, the *RSoC framework* must support these tools.

Altera provides Qsys system integration tool (part of Quartus design environment) and Xilinx formerly provided the Embedded Development Kit, today replaced with IP integrator (part of Vivado design environment).

The purpose of these tools is to let user to quickly create a system based on either a hard processor (ARM) or soft core (such as *Microblaze* or *Nios*). The system is integrated with help of a graphical interface in which user can create a system from available *IP* cores simply just by clicking a mouse. This is illustrated on the Figure 3.16.

Apart from user interface differences these tools provides different *APIs* and supports different interfaces.

Nevertheless *Qsys* supports all the necessary interfaces for *RSoC Bridge* (*AXI*, *AXI*4 *Lite* and *AXI Stream*).

![](_page_31_Figure_0.jpeg)

Qsys - soc\_system.qsys\* (/home/bruno/VUT\_FIT/WORK/IOT\_WORK/socfpga\_test\_new\_wrapper/soc\_system.qsys) - + ×
 Edit System Generate View Tools Help

Figure 3.16: Example of a system in Altera QSys

#### 3.5 Drivers

This section will present a brief overlook of drivers (known as RSoC Driver) developed over the course of the time when the RSoC Bridge was ported. These drivers were all originally implemented for the Xilinx platform and then ported over to Altera.

#### 3.5.1 Standard Linux driver

First driver ported over to *Altera* platform. It is a *Linux* based driver with standard character device *API* (read() and write()) for reads and writes.

The drawback of this driver is that the data is always copied between user-space buffer and kernel-space buffer.

Another drawback is that the low-level logic which controls the DMA controller is coupled with higher-level OS specific memory handling and Linux character device implementation.

#### 3.5.2 Zerocopy RX Linux driver

To address the copy issue of previous driver, the Zerocopy driver was introduced. Driver was developed only for upstream direction.

However the the Zerocopy driver has a nonstandard API (for a character device). The user-space application is far more complex opposing to a simple read() and write(). The application must handle the buffers associated with descriptors directly.

The advantage is that copying is in fact measured (shown later in the Section 5.3.1) to have one of the greatest impact on performance of the data transfer.

#### 3.5.3 Multiplatform driver

To make the *RSoC Bridge* available on the *FreeRTOS* platform a multiplatform driver was developed. This driver also addresses the second issue of the standard driver by separating low-level driver logic from high-level OS specific logic (the zerocopy capability is also preserved in this driver).

Since this driver is currently perspective for the future development of the RSoC Bridge we will cover it in more detail than previous drivers and finally we will demonstrate the port of the driver between Xilinx and Altera platform on this driver.

The driver specification [12] specifies that the communication between the high-level layer and low-level layer is done through a ring of descriptors. A ring defines two pointers - *head* and *tail*.

- The *head* pointer is always managed by the source of data (in case of RX it is low-level layer, in case of TX it is a high-level layer).
- The *tail* pointer is always managed by the consumer of data (in case of RX it is high-level layer, in case of TX it is a low-level layer).

The high-level layer is dependent on the target operating system, for example in case of *Linux* it is implemented as kernel module which maps the character device functions to the low-level *API*.

The logic of high-level layer expect following semantics of commit() and poll() functions implemented in low-level logic of the driver.

#### **RX Semantics**

- rx\_commit(): updates hardware with *pre-tail* (if needed) which will allow a *DMA* controller to fill the ring with maximum amount of data.
- rx\_poll(): moves head for the high-level if there are completed transactions

The following algorithm is executed by high-level layer:

```
rx_poll()
while(True):
    read_data_from_ring
    move_tail
rx_commit()
```

Listing 3.1: High-level layer RX algorithm

#### **TX** Semantics

- tx\_commit(): updates hardware with *head* which will start the transfer
- tx\_poll(): moves *tail* for the high-level if there are completed transactions

The following algorithm is executed by high-level layer:

```
tx_poll()
while(True):
    write_data_to_ring
    move_head
tx_commit()
```

Listing 3.2: High-level layer TX algorithm

## Chapter 4

## Design and implementation

In this chapter we will propose a solution and deal with the design of the actual implementation of port of the RSoC Framework to the Altera SoC platform. First we will introduce the FPGA components used, then we will cover the integration of the framework to Altera design tools and finally we will cover the necessary software drivers for the Altera version of the RSoC Bridge.

#### 4.1 **Proposed solution**

From the architectural and software differences of *Xilinx* and *Altera* platforms we can see that critical part is a port of the *DMA controller*.

There are few different possible solutions:

- Implement platform independent S & G DMA controller based on AXI.
- Modify third party platform independent S & G DMA controller to use AXI.
- Modify the Altera S & G DMA controller to use AXI interface.
- Add AXI Avalon adaptors to the Altera S&G DMA controller.
- Modify *RSoC Bridge* to use Avalon instead of *AXI*.

From the future-wise point of view it would be beneficial to implement an own platform independent  $S \mathscr{C} G DMA$  or a use third-party platform independent  $S \mathscr{C} G DMA$  controller. This would allow port to yet another platform to be independent on vendor's DMA controller.

But implementing a custom S & G DMA controller would be time consuming and error prone and no easily adaptable to AXI platform independent third-party S & G DMA is available.

Modifying the RSoC Bridge to use Avalon instead of AXI would bring incompatibility between the RSoC Bridge on Xilinx and Altera platforms and that is undesirable.

And since modification of the Altera S & G DMA is prohibited by the license and Altera is constantly updating their IPs, it has been decided that the best solution from practical point of view is to create AXI - Avalon adaptors and use the Altera S & G DMA controller.

The port can be therefore basically divided into these four steps:

1. Design and develop AXI — Avalon adaptors. Integrate the Altera S & G DMA with the RSoC Framework DMA entities using the developed adapters.

- 2. Create a platform dependent integration tool scripts and a top level design.
- 3. Modify RSoC Driver to work with the Altera  $S \otimes G$  DMA controller.
- 4. Test function and measure the performance of the RSoC Framework on the Altera Cyclone V SoC platform.

#### 4.2 Components for integration of the Altera S&G DMA to the RSoC Bridge

The following set of components integrates the Altera S & G DMA to the RSoC Bridge. These components are used in the RSoC DMA wrapper component for the Altera S & G DMA with the interface specified by the RSoC Bridge.

#### 4.2.1 RSoC DMA wrapper for the Altera S&G DMA

The Altera S & G DMA is integrated to the RSoC Framework utilising a wrapper component. The architecture of the wrapper for Altera S & G DMA to use with the RSoC Framework interface can be seen on the Figure 4.1. As it has been covered in the Section

![](_page_35_Figure_7.jpeg)

Figure 4.1: Architecture of the dma\_ctrl\_altera

3.3 the architecture allows to enable upstream or downstream only or both. Therefore

the dma\_ctrl\_altera component consists of two Altera S & G DMA controllers, one for *upstream* and one for *downstream*. These controllers has three interfaces:

- Memory access Avalon-MM master.
- Descriptor (S&G) access Avalon-MM master.
- Control Status Avalon-MM (non-bursting) slave.

These are connected to respective AXI Slaves: Read-write access to descriptors is connected through an Avalon-MM to AXI4-Full adaptor and then through an AXI 2-to-1 (merger) to the  $M\_SG$  AXI4-Full slave interface. Upstream write only and downstream read only memory access is connected through a common Avalon-MM to AXI4-Full adaptor.

Since user can enable only *upstream* or *downstream* for their application, then only one of the Altera S&G DMA controllers is used. Therefore only one Avalon-MM to AXI4-Full adaptor is connected to the  $M_SG$  AXI4-Full slave interface (the AXI 2-to-1 (merger) is not used).

Finally the AXI Stream slave interface is converted to the Avalon-ST master interface for the upstream DMA controller. Similarly the AXI Stream master interface is converted to the Avalon-ST slave interface for the downstream DMA controller.

The soft reset controller is used to reset the axis\_sof unit inside the stream adaptors. This is necessary to synchronise the axis\_sof unit when DMA controller is software reset (the DMA controller would not consume stream when no *SOF* signal is generated).

End of packet issue During the development of the driver there was an issue with the Altera S & G DMA controller. The controller does not stop on end of packet signal when descriptor is configured with predefined bytes to transfer. The new Altera modular S & G DMA controller (as described in the Section 3.3.1) does not have this issue, but it was not available when the initial design consideration have been made. This has direct consequence on the architecture of the dma\_ctrl\_altera by adding a Stream slicer to upstream as shown on the Figure 4.1.

The end of packet signal is used by the controller only when a descriptor is configured to *infinite bytes to transfer* (when *bytes to transfer* is set to 0). This is not a problem in an application where there is known maximal length of the frame such as the *ethernet*. In case of the RSoC Bridge this does pose a problem since we can not assume maximal length of frame.

To solve this issue an *end of packet* signal must be issued before a given number of words is transferred. This is carried out by the axis\_slicer component where the driver can set the number of words to transfer.

In the current implementation, this component utilises a simple counter that will always issue *end of packet* when a given length of words is transferred. However this can lead to suboptimal frame sizes.

Another possible solution is to merge all frames except of the last frame and slice them to optimal frame size. This implementation would be substantially more complex (and would increase the logic utilisation), since words can be unaligned and the unit can not know when the last frame occurs (some timeout logic would have to be present).

#### 4.2.2 AXI — Avalon adaptors

As described in the previous section, to adapt the *Altera S&G DMA* controller to the *RSoC Bridge* we need to adapt *Altera* specific interfaces to the *AXI* counterparts. The following sections describes each adaptor component needed in the wrapper.

#### Avalon-MM (pipelined, bursting) to AXI4-Full

The Avalon-MM (pipelined, bursting) to  $AXI_4$ -Full converts the master Avalon-MM interface (with burst capability) to the  $AXI_4$ -Full interface. This interface is utilised by the DMA controller for memory access and therefore maximising throughput is the key in this component.

![](_page_37_Figure_4.jpeg)

Figure 4.2: Avalon-MM to AXI adaptor

The Figure 4.2 shows the basic architecture of the Avalon-MM to AXI4-Full adaptor. This adaptor consist of read and write channel adaptor (Avalon-AXI Read and Avalon-AXI Write).

The read channel shown on the Figure 4.3 buffers outstanding read requests as the ARFSM controls the FIFO storing Avalon-MM read requests (the read request stored in the FIFO consists of the burst length and read address). While the RFSM executes these buffered transactions on AXI R channel.

The Figure 4.4 and the Figure 4.5 shows the state transitions of the respective state machines.

![](_page_38_Figure_0.jpeg)

Figure 4.3: Avalon-AXI Read

![](_page_38_Figure_2.jpeg)

Figure 4.4: State machine for buffering Avalon read transactions (ARFSM)

![](_page_39_Figure_0.jpeg)

Figure 4.5: State machine acting as AXI transactor (RFSM)

![](_page_40_Figure_0.jpeg)

Figure 4.6: Avalon-AXI Write

![](_page_40_Figure_2.jpeg)

![](_page_40_Figure_3.jpeg)

The write channel is very similar, but significantly simpler as can be seen on the Figure 4.6, since Avalon-MM does not support outstanding transactions for write channel.

No FIFO is needed - a register is used. Only one FSM is used (*WFSM* shown on the Figure 4.7) for the write channel adaptor. The write counter is used to determine when to generate *WLAST* signal (which is not used on the *Avalon-MM* interface and must be determined from the burst length).

#### AXI4-Lite to Avalon-MM (non-pipelined, non-bursting)

This unit converts AXI4-Lite to Avalon-MM with a chipselect vector, so that multiple Avalon slaves can be accessed from single adaptor. The address space of Avalon slaves can

![](_page_41_Figure_0.jpeg)

Figure 4.8: AXI4Lite to Avalon-MM

be user defined. The adaptor shown on the Figure 4.8 consist of AXI Lite Write and AXI Lite Read endpoints. These units solves AXI transaction handling.

Arbiter selects which of the two channel will be translated to Avalon (it only supports either write or read at the single moment), address selects write/read address according to result of arbitration (basic Avalon interface shares the address and byteenable signals for read and write). Decoder selects device by its address and address rebase decrements device address prefix - so that target device gets addresses that starts from address 0.

The Wait generator is used to generate *waitrequest* when fixed Wait-States are enabled. When they are disabled, this component is bypassed (and not generated) and *waitrequest* input is used directly.

**AXI Lite READ** The AXI Lite READ endpoint provides a simple register access interface. As shown on the Figure 4.9 this component consists of a *FSM* and *data-path*.

The semantics of provided signals is as follows. When the read transaction occurs on the  $AXI \ AR$  channel the FSM asserts  $READ \ REQ$  which indicates request for read (at the same time  $READ \ ADDR$  is set to latched  $AXI \ ARADDR$ ). The connected component then should provide the data on  $READ \ DATA$  signal and acknowledge read by asserting  $READ \ ACK$ . The FSM then sends data and response through  $AXI \ R$  channel.

![](_page_42_Figure_0.jpeg)

Figure 4.9: AXI Lite READ

**AXI Lite WRITE** The AXI Lite WRITE endpoint provides a similar interface as READ counterpart. As show on the Figure 4.10 the *FSM* and *data-path* is also present here.

The semantics is also similar. When the write transaction occurs on the AXI AW channel the FSM asserts WRITE REQ (with address). The FSM latches WDATA to the WRITE DATA (the strobe - byteenable is also latched) and waits until the connected component acknowledges the write by asserting WRITE ACK. The FSM then sends respond on AXI B channel.

![](_page_42_Figure_4.jpeg)

Figure 4.10: AXI Lite WRITE

#### AXI4-Stream to Avalon-ST and Avalon-ST to AXI4-Stream

AXI4-Stream and Avalon-ST are almost a matching interface (with different naming of signals). Except key difference on keep and empty signals where:

- AXI4-Stream Keep: Indicates whether associated byte is a part of the data stream.
- Avalon-ST Empty: Indicates the number of symbols that are empty during cycles that contain the end of a packet.

The empty\_keep and keep\_empty components of respective adaptors does this conversion. The startofpacket signal is generated by axis\_sof component when converting from AXI4-Stream to Avalon-ST. Other signals maps directly according to the Table 4.1. The internal structure of the adaptors is shown on Figures 4.11 and 4.12.

AXI-Stream	Avalon-ST
TID	CHANNEL
TDEST	no connect
TDATA	DATA
TUSER	ERROR
TLAST	ENDOFPACKET
TREADY	READY
TVALID	VALID
no connect	STARTOFPACKET

Table 4.1: Avalon-ST and AXI4 Stream assigned signal

![](_page_43_Figure_5.jpeg)

Figure 4.11: AXI4-Stream to Avalon-ST Figure 4.12: Avalon-ST to AXI4-Stream

ready validt

tready tvalid SNK\_AV

error

channel

tid

0

empt

data

tdata

keep

empty\_

axis\_reg

ckeep

tuser tdest

VM\_AXI

**empty\_keep component** The empty\_keep component is implemented with offset\_keep where offset = length(keep) - 1 - empty. The algorithm of offset\_keep conversion is shown on the Listing 4.1.

```
def offset_keep(offset, keep_width):
    keep[0:keep_width]
    for i in range(len(keep)):
        if i <= offset:
            keep[i] = 1</pre>
```

return keep

Listing 4.1: offset\_keep algorithm

**keep\_empty component** The empty\_keep component is implemented with keep\_offset where empty = length(keep) - 1 - offset. The algorithm of keep\_offset conversion is shown on the Listing 4.2.

```
def keep_offset(keep):
    for i in range(len(keep)):
        if keep == offset_keep(i, len(keep)):
            offset = i
            break
    return offset
        Listing 4.2: keep_offset algorithm
```

#### 4.3 Integration with Altera tools

To use the RSoC Framework practically on the Altera SoC platform, the RSoC Framework must be integrated with Altera design tools. This ranges from synthesis support (e.g. rewriting Xilinx specific parts of the HDL for Altera, creating a top-level design) to support of graphical system integration tools (Qsys in case of Altera).

And finally, also the *IP* core distribution must be solved since the *RSoC Framework* is proprietary system and it is not desirable to distribute it in the source code form. This also allows distribution of trial versions and other limitations of the usage.

#### 4.3.1 Altera Quartus synthesis support

Several modifications had to be performed to support synthesis of the RSoC Framework under the Altera Quartus. Among them the most significant modifications were creating a string based configuration system and creating a wrapper design for the Altera Quartus.

**String based configuration system** The RSoC Framework has a synthesis time configuration system based on configuration file. This system simplifies the configuration of the RSoC Bridge by reducing the number of used generic VHDL parameters and reading the necessary parameters from configuration file instead.

However the *Altera Quartus* synthesiser does not support reading files in *VHDL* during the synthesis. This breaks the *RSoC Bridge* synthesis. To overcome this issue and still have the ability to use single configuration file a String based configuration system was created.

This system reads a single string generic instead of a file (it is actually using the VHDL generic used originally for the configuration file name). To integrate this system a wrapper design was written in *Verilog* so that this constant is included as a *Verilog* header file which acts as original configuration file still retaining the simplicity of original file based configuration system. The only difference is that the configuration file is not a plain text file, but it has definition of a *Verilog* macro (which contains the configuration as single string) instead.

```
'ifndef _altera_base_include_
                                           'define _altera_base_include_
[general]
                                           'define CONSTANT BASE "\
C_FAMILY = Zynq
                                           [general]; \
[bridge-sbus]
                                           C_FAMILY = Cyclone V; \setminus
master.1.protocol = AXI4
                                           [bridge-sbus]; \
                                           master.1.protocol = AXI4; \
master.1.baseaddr = 0
master.0.id_width = 12
                                           master.1.baseaddr = 0; \setminus
master.0.highaddr = 0
                                           master.0.id_width = 12; \setminus
                                           master.0.highaddr = 0; \setminus
Listing 4.3: Xilinx configuration file ex-
                                           `endif
ample
                                           Listing 4.4: Altera configuration file ex-
                                           ample
```

On the listings 4.3 and 4.4 we can see that the configuration for *Altera* can be easily generated from the original *Xilinx* by adding a constant *Verilog* header and line separators (since it is a single string in the result, we need the original end of line information for parsing the configuration).

**Top-level design** The RSoC Bridge itself is generic to configuration of the buses available on the target platform. This means that the RSoC Bridge has only generic ABUS, MBUS, SBUS and interrupt signal vectors. The wrapper design must do the mapping of the generic vectors to the concrete interfaces and it must make sure that for the set configuration the connection to these generic interfaces matches the configuration of the RSoC Bridge. This wrapper (shown on the Figure 4.13) is similar to the original Xilinx implementation, but the interface naming is according to Altera.

![](_page_45_Figure_3.jpeg)

Figure 4.13: Top-level design overview

In addition a Verilog wrapper was created to support the String based configuration

system and simplify the mapping of interrupts.

#### 4.3.2 Altera Qsys system integration tool support

The importance of the integration with the *Altera Qsys* lies in the fact, that this part of the RSoC Bridge is a user interface visible to the end user of the RSoC Bridge. The key is to make the usage of the RSoC Bridge as convenient as possible while maintaining the ability to configure the aspects of the RSoC Bridge system.

The original *Xilinx* implementation of the user interface for the *Qsys* counterpart (Xilinx EDK) let the user to configure every aspect of the *RSoC Bridge*. This lead to an arguably complex configuration and convoluted user interface.

rsoc_bridge_socfpga - rsoc_bridge_socfpga_0 x				
rsoc_bridge_socfpga	<u>D</u> ocumentation			
show signals      Show signals      soc_bridge_socfpga_0      clock_sink     clock     clock clock     a0_c      clock_sink     clock     clo	Image: Common         Count of Accelerators:         Image: Common         Count of user events:         Count of user events:         Configuration bus used by the accelerator:         HPS-to-FPGA         Address range of configuration bus:         S&G and Data width of DMA:         S&G & Data association:         FPCA-to-HPS SDRAMO         Image: Common         Image: Common         Image: Common         Image: Common         S&G & Data association:         FPCA-to-HPS SDRAMO         Image: Common         Image: Common			
Warning: rsoc_bridge_socfpga_0: 32bit ACP (HPS-to-FPGA) is forced for accelerator 0, because combination of 32 and 64bit DMA interfaces is used Info: rsoc_bridge_socfpga_0: Number of accelerator assigned to H2F: 2 Info: rsoc_bridge_socfpga_0: Number of accelerator assigned to H2F_LW: 1 Info: rsoc_bridge_socfpga_0: HPS-to-FPGA address width is: 13.0 Info: rsoc_bridge_socfpga_0: HPS-to-FPGA Lightweigth address width is: 6.0				

Figure 4.14: Simple configuration of the RSoC Bridge in Qsys

When developing the *Altera* version the configuration were automated by merging various aspects of the *RSoC Bridge* into a single drop down menu. And also configuration presets (a quick configuration) were introduced, this allows to create configuration presets stored in files and let user chose from these preset to have quick access to commonly used configurations. This is shown on the Figure 4.14.

These presets also help with the *IP* core distribution, especially when it is done in the *netlist* form, because it creates a standard set of configurations.

#### 4.3.3 IP core distribution

An unavoidable part of the *Altera* port of the RSoC Bridge is its distribution. To protect the intellectual property it is desirable to not distribute the source codes of the RSoC

#### Bridge.

To solve this task on the *Xilinx* platform, the distribution is done through synthesised *netlists* in the *EDIF* format. This solution only protect source codes, it cannot prevent unauthorised redistribution of the *IP* itself.

However on the *Altera* platform there is no option to use synthesised *netlists*, since the *Altera Quartus* can not generate *netlists* anymore. For the *Altera* platform following two possible solutions exists of each has its own advantages and disadvantages.

#### **Design** partitioning

The primary role of the design partitioning is to separate the design into functional blocks. Partitioning is used to not only logically separate modules, but often to also separate them physically on the die. This is used for better team cooperation and also to help synthesis tools.

This is done especially in large (FPGA or ASIC) designs to have faster synthesis and easier routing of the design by allowing the synthesis to work only on modules being modified, while others are not re-synthesised. This allows to have pre-synthesised modules which can be used for the IP core distribution.

Main disadvantage of this approach is that *Design partitioning* is not a part of free license of the *Altera Quartus*. That would force any user of the *IP* distributed as a design partition to obtain a full license of the *Altera Quartus*.

Another disadvantage is that it pose an additional effort for the user to set up the design partition. Also since this design partition is pre-synthesised, for every configuration of the *IP* core there would have to be a matching pre-synthesised design partition file (but this also applies to the *netlist* distribution).

The advantage of this approach is obviously the fact, that the design is already synthesised, that means the synthesis is fairly faster than with source codes, especially since the RSoC Bridge is highly generic.

#### Altera design files encryption

A complete different approach is to encrypt the source *HDL* design files. This is supported by the *Altera AMPCRYPT* tool which is distributed to *Altera* partners. This tools allows to encrypt source files and distribute this encrypted files with a license with different permissions (e.g. license can allow only simulation).

The main advantage of this approach is that the IP core is in the source form, so it is configurable. This allows to distribute only one IP core and let user configure it. Disadvantage of this approach is that AMPCRYPT is available only upon request to *Altera* partners.

**Fixed configurations** It is not always desirable to allow user to configure the *IP*. To limit the set of available configurations, a fixed string based configuration were introduced. This allows to inject only a set of configurations and encrypt them with the sources of the design. Then the user can select configuration only from the limited list of configurations (but still use the same design files).

#### 4.4 Drivers

To complete the port of the RSoC Bridge it was necessary to modify drivers for the Altera S & G DMA. The key differences between Altera and Xilinx DMA controllers were overviewed in the section 3.3.

The implementation of the *Altera* drivers follows the *Xilinx* implementation when regarding memory management and cache coherency and device discovery. The main difference is the management of the hardware and descriptor processing.

#### 4.4.1 Descriptor processing differences between Altera and Xilinx drivers

The descriptor processing was described in the section 3.3 from hardware point of view. Now we will describe differences in the descriptor processing from the software point of view - while we will focus only on implementations of the low-level parts of the *multiplatform* driver as described in the Section 4.4. This is done for clarity, since the *multiplatform* driver has a relatively simple low-level interface.

As you can see in the Table 4.2 there are few differences in fields of the descriptors the that needs to be addressed when porting a driver logic from *Xilinx* to *Altera*. The most significant one is that *Altera* does not stop processing on the *end of packet* when the *bytes* to transfer field is set.

The solution to this issue was described in the Section 4.2.1. From the software point of view it means that *upstream* part of the driver has to set word count to the *stream slicer* instead of *bytes to transfer* field. This also means that driver can not obtain the original frame boundaries.

Semantics	Altera	Xilinx
Address of read buffer	read address	phy
Address of write buffer	write address	phy
Address of next descriptor	next descriptor	next
	pointer	
Number of bytes to transfer	bytes to transfer	control   buffer
	-	length
Length of read burst	read burst	
Length of write burst	write burst	
Actual bytes transferred	bytes transferred	status   transferred
		bytes
Owned by hw/complete	control   owned by	status   Cmplt
- , -	hw	
Generate EOP/EOF	control   generate	control   TXEOF
,	еор	·
Generate SOP/EOF	control   write	control   TXSOF
,	fixed	
Contains start of frame		status   RXSOF
Contains end of frame		status   RXEOF
5 5		·

Table 4.2: Table of semantic differences between descriptors on Altera and Xilinx DMA

Other fields have similar semantics, but their naming is different or values are inverted. One significant thing to note is the relation of the *complete* flag in *Xilinx* and *owned by*  hardware flag in Altera. The owned by hardware flag of Altera can mark complete descriptors in a combination with bytes transferred.

The other use of the owned by hardware flag is for the DMA controller to know when to stop processing (there are no free buffers left). This semantics is carried out by the tail descriptor pointer in the Xilinx DMA controller instead (which is not field of the descriptor but rather field in control registers of the DMA controller). This actually makes the descriptor processing simpler on the Altera platform, since the driver does not need to update the tail descriptor pointer. Only when the Altera DMA controller stops on the descriptor that has owned by hardware flag set to 0 it must set next descriptor to process and start the DMA controller again.

These differences are best illustrated on the following pseudocode examples of commit and poll implemented for the low-level part of the *multiplatform* driver.

```
def commit_rx():
    pretail_desc = get_ring_pretail() # descriptor just before tail
    controller.rx_tail = pretail_desc
    return 0
```

Listing 4.5: Xilinx commit\_rx()

```
def commit_rx():
    if dma_is_busy():
        return 0
        controller_rx.stop()
        controller_rx.next_descriptor = get_ring_head()
        controller_rx.start()
        return 0
    }
}
```

Listing 4.6: Altera commit\_rx()

Note that Altera RX commit does not update anything in the DMA controller when the controller is running as opposed to Xilinx RX commit.

The situation is similar in case of TX commit, while Xilinx one (shown on Listing 4.7) actually sets the Xilinx DMA controller tail to software pre-head this is not done in the Altera port, only when the controller stops the next descriptor is set to ring tail and the controller is reset. (Listing 4.8).

```
def commit_tx():
    prehead_desc = get_ring_prehead() # descriptor just before head
    controller.tx_tail = prehead_desc
    return 0
```

Listing 4.7: Xilinx commit\_tx()

```
def commit_rx():
    if dma_is_busy():
        return 0
        controller_tx.stop()
        controller_rx.next_descriptor = get_ring_tail()
        controller_tx.start()
        return 0
        Listing to the second sec
```

Listing 4.8: Altera commit\_tx()

## Chapter 5

## Testing and measurements

This chapter presents the methods to test and measure the performance of the RSoC Bridge on the Altera platform. Simulation and testing of component function is described as well. There is also a brief overlook of the used hardware platform.

#### 5.1 Hardware platform

The primary development platforms were *EBV Socrates* and *Altera Cyclone V SOC De*velopment Board. These platforms will be briefly introduced in following paragraphs.

**EBV Socrates** *EBV Elektronik* advertises SoCrates as a low cost Cyclone V SoC starter kit [13].

![](_page_50_Picture_6.jpeg)

Figure 5.1: EBV Socrates Starter Kit [13]

It is loaded with the following key components:

- Altera Cyclone V SoC device 5CSEBA6U23C7N (110K LEs) the used kit had engineering sample device populated.
- 128\*32Mbit (512MB) DDR3 memory.
- 1Gbit Ethernet, CAN, USB 2.0 OTG.

- Micro-SD Card Slot.
- Embedded USB Blaster II (implemented using an Altera MAX CPLD).

Altera Cyclone V SOC Development Board The Altera board is a more comprehensive development kit [3].

![](_page_51_Picture_3.jpeg)

Figure 5.2: Altera Cyclone V SOC Development Board [3]

It features following components:

- Altera Cyclone V SX SoC device 5CSXFC6D6F31C8NES (110K LEs), this device features high speed transcievers (PCIe, SATA..).
- Altera MAX V CPLD device 5M2210ZF256C4N (system controller).
- MAX II CPLD device EPM570GF100 (embedded USB-Blaster II).
- 1GB DDR3 SDRAM (32 bit) for FPGA.
- 1GB DDR3 SDRAM (32 bit) for HPS (ECC).
- 1Gbit Ethernet, CAN, USB 2.0 OTG.
- 2X 10/100 Ethernet PHYs (EtherCAT).
- PCIe 1 x4 slot.
- Universal high-speed mezzanine card slot (HSMC).
- Micro-SD Card Slot.
- SMA input for HPS clock.
- Linear Technology System monitoring circuit (voltage, current, power).

#### 5.2 Simulation and unit tests

To test developed components unit tests were used. The drawback of the unit test in this case is that it is hard to simulate the behaviour of some components.

Since some of the components used are proprietary, it is not always possible to determine their behaviour in some situations. Therefore testing such a component is not an easy task. An example is the *Altera S&G DMA* controller. The behaviour of its *control and status register* interface was not clear and although the unit tests for this interface passed (written according to the *Altera* specifications) the system was not working correctly.

To test these components properly, a more complex integration test was used which included simulation of the *Altera DMA* engine.

Another example of such peculiar component is the *Hard Processor System* itself. The behaviour of this component is determined by the software running in the ARM processor, therefore it is really difficult to simulate it on the RTL level.

The used approach was therefore to use simple unit tests and some integration tests and then test the system in the hardware. When some issue has been found the unit test or integration was extended to test for such issue. The issue was then solved using this extended test. This is a lengthy process since every change in the hardware design means that whole system must be synthesised.

The advantage of these unit tests is that there is simple check for regressions when there are any modifications to the code. These test can also run automatically (for example scheduled to run over a night) to test if any changes have not broken some functionality. And when they incorporates tests for issues that had been found during tests in hardware, it makes sure that these issues will not occur again.

#### 5.3 Performance testing and measurements

The performance of the *RSoC Bridge* on the new platform must be measured to make sure that no unnecessary and additional bottlenecks were introduced in the process of porting and also to test the correct operation of the implemented components and drivers.

This measurement is also important to get the data to know which use cases can be solved by the RSoC Bridge on the particular platform. The same goes for the estimations of the resource utilisation.

#### 5.3.1 Throughput measurements

All measurements were performed on 50 MHz design. The kernel buffers were set in range of  $4 \ kB \ to \ 32 \ kB$ , the userspace buffer size followed the size of kernel buffer in all measurements.

#### Zerocopy RX driver

Mossuromont	ovplicit sync of cachos	dma alloc cohoront
	explicit sync of caches	uma_anoc_concrent
4 KB		
Read and ack descriptors	222  MB/s	222  MB/s
Do vmsplice to /dev/null	180  MB/s	222  MB/s
Access every data word	130  MB/s	30  MB/s
8 kB		
Read and ack descriptors	224  MB/s	224  MB/s
Do vmsplice to /dev/null	172  MB/s	222  MB/s
Access every data word	134  MB/s	$30 \mathrm{~MB/s}$
16 kB		
Read and ack descriptors	225  MB/s	$225 \mathrm{~MB/s}$
Do vmsplice to /dev/null	168  MB/s	225  MB/s
Access every data word	$137 \ \mathrm{MB/s}$	$30 \mathrm{~MB/s}$
32 kB		
Read and ack descriptors	$225 \mathrm{~MB/s}$	$226 \mathrm{~MB/s}$
Do vmsplice to /dev/null	166  MB/s	226  MB/s
Access every data word	$140 \ \mathrm{MB/s}$	$30 \ /\mathrm{MB/s}$

Table 5.1: Performance of the Zerocopy driver on Altera

Measurement	explicit sync of caches	$dma_alloc_coherent$
4 kB		
Read and ack descriptors	$210 \mathrm{~MB/s}$	$276 \ \mathrm{MB/s}$
Do vmsplice to /dev/null	145  MB/s	276  MB/s
Access every data word	119  MB/s	$30 \mathrm{~MB/s}$
8 kB		
Read and ack descriptors	229  MB/s	276  MB/s
Do vmsplice to /dev/null	146  MB/s	276  MB/s
Access every data word	125  MB/s	30  MB/s
16 kB		
Read and ack descriptors	$240 \ \mathrm{MB/s}$	348  MB/s
Do vmsplice to /dev/null	170  MB/s	348  MB/s
Access every data word	130  MB/s	30  MB/s
32 kB		
Read and ack descriptors	$246 \ \mathrm{MB/s}$	364  MB/s
Do vmsplice to /dev/null	176  MB/s	364  MB/s
Access every data word	132  MB/s	$30 \ /\mathrm{MB/s}$

Table 5.2: Performance of the Zerocopy driver on Xilinx

#### Multiplatform driver

The performance of the multiplatform driver was tested with the standard blocking read and write (which in turn use copies between the userspace and the kernel space). The reading and writing processes were not locked on the CPU core.

Measurements were done on the Linux 3.15.0 in the case of *Altera* and Linux 3.17.0 in the case of *Xilinx*.

The actual measured performance can be seen on the Table 5.3 and 5.4.

Kernel	RX	TX	Loopback
buffer			
size			
4k	$149 \mathrm{~MB/s}$	$138 \mathrm{~MB/s}$	$34 \mathrm{MB/s}$
8k	$165 \ \mathrm{MB/s}$	$120 \ \mathrm{MB/s}$	$102 \ \mathrm{MB/s}$
16k	$174 \mathrm{~MB/s}$	$120 \ \mathrm{MB/s}$	$114 \mathrm{~MB/s}$
32k	$177 \; \mathrm{MB/s}$	$147 \mathrm{~MB/s}$	109  MB/s

Table 5.3: Performance of the multiplatform driver on Altera

Kernel	$\mathbf{R}\mathbf{X}$	TX	Loopback
buffer			
size			
4k	$146 \mathrm{~MB/s}$	$173 \ \mathrm{MB/s}$	32  MB/s

Table 5.4: Performance of the multiplatform driver on Xilinx

**Perf analysis of the multiplatform driver** The perf analysis of the multiplatform driver shows that in fact the copy between the userspace and kernel space is one of the main bottlenecks in the current driver implementation.

32.80%	[kernel]	[k]	v7_dma_inv_range
23.22%	[kernel]	[k]	copy_to_user_std
22.03%	[kernel]	[k]	_raw_spin_unlock_irqrestore
2.89%	[rsocdrv]	[k]	hwring_ctrl_read
1.95%	[rsocdrv]	[k]	Util_ioread32
1.54%	[rsocdrv]	[k]	rx_poll
1.51%	[kernel]	[k]	apbt_read_clocksource
1.46%	[kernel]	[k]	vector_swi
0.66%	[kernel]	[k]	ktime_get_ts
0.59%	[kernel]	[k]	fsnotify
0.56%	[kernel]	[k]	arm_dma_sync_single_for_cpu
0.52%	[kernel]	[k]	vfs_read
0.44%	[kernel]	[k]	fget_light
0.42%	libc-2.18.so	[.]	0x000e10f8
0.41%	[kernel]	[k]	ret_fast_syscall
0.41%	[kernel]	[k]	sys_clock_gettime
0.40%	[rsocdrv]	[k]	Util_iowrite32
0.38%	[kernel]	[k]	sys_write

0.35%	[kernel]	[k]	fdget_pos
0.34%	[kernel]	[k]	finish_task_switch

Listing 5.1: Perf analysis of the multiplatform driver

#### 5.3.2 Resource utilisation

The following tables shows a brief comparison of the resource utilisation on the *Altera* and *Xilinx* platform. The *Xilinx* part is a 85K Logic Cells device and *Altera* part is a 110K Logic Elements, which should be comparable numbers according to *Altera* claims (discussed in the Chapter 3).

Site Type	Used	Available	Util%
Logic Utilization	5323	83820	6
ALUTs	2073	83820	2
Dedicated logic registers	4755	167640	2
ALMs partially or completely used	2856	41910	6
Total LABs: partially or completely used	294	4191	7

Table 5.5: RSoc Bridge generic core with one duplex DMA controller(1x accelerator, 64b data, F2H SDRAM0, F2H, Altera 5CSEBA6U23C7N)

Site Type	Used	Available	Util%
Slice LUTs	4727	53200	8.88
Register as Flip Flop	6141	106400	5.77
BRAM38/FIFO	9	140	6.42
BRAM18	2	280	0.71

Table 5.6: RSoC Bridge generic core with one duplex DMA controller (1x accelerator, 64b data, HP0, Xilinx xc7z020)

Even though the compared devices are not the same density. The comparison on Tables 5.3.2 and 5.3.2 shows that the *RSoC Bridge* has a fairly similar logic utilisation in similar configuration on these devices.

## Chapter 6

## Conclusion

The widespread usage of System-on-Chip solutions led programmable logic manufacturers to develop reconfigurable version of these chips. When Xilinx introduced their Zynq platform many applications emerged for this platform. To simplify development of these applications the RSoC Framework was developed for the Xilinx Zynq platform.

In this work we have explained what is a *System-on-Chip*, reasons to use it in an embedded system and the advantages of a reconfiguration.

Further explanation shown currently available solutions for both Xilinx and Altera platform. Then introduced the RSoC Framework and its advantages over currently available frameworks or development from the scratch. This led to decision to port this framework over to the Altera platform.

Then we have covered the key architectural details of the *RSoC Framework*. The essential hardware and software details of the *Altera* and *Xilinx* platforms and pointed out the key differences between these platforms.

Later on we have proposed a solution to port the RSoC Bridge over to the Altera platform. The implementation of the proposed solution is then explained through the FPGA firmware component design to the port of the drivers.

The work finally presents a successful port of the RSoC Bridge on the Altera SoC platform. This is demonstrated in the Chapter 5 Testing and measurements, where the performance matches the performance of the Xilinx implementation. The resource utilisation is also comparable. The Altera SoC platform has a widespread use, in fact the market share is competitive to that of the Xilinx Zynq platform. This makes the RSoC Bridge interesting for the users of the Altera programmable logic.

Further work could focus on the integration of the new components and support of the new features by *Altera*. It would be also advantageous to implement a platform independent components for the *RSoC Bridge*, this is especially true for the *DMA* controller. The platform independent *DMA* controller would unify drivers across different platforms. If the other manufacturers of the programmable logic devices would introduce the *FPGA* based *SoC* on these platforms could be targeted as well without much effort.

The two biggest manufacturers — Altera and Xilinx also introduced the new line of their FPGA based System-on-Chip. The Zynq UltraScale+ MPSoC by Xilinx and Stratix 10 SoC by Altera. The support for these chips will be certainly needed.

An ideal way to simulate these kinds of systems would be to simulate the *hard processor* in some cycle accurate CPU emulator such as Qemu and drive the inputs of HDL simulation (such as *Modelsim*) from this emulator. This would greatly help users in development of applications for the framework and also of the framework itself. The future work could

therefore focus on the development of such simulation system.

Other work could focus on further testing of the RSoC Bridge under different conditions. The functional verification of the implemented components and the RSoC Bridge as whole would add the confidence for the applications of the RSoC Framework in the industry.

## Bibliography

- [1] 7 Series FPGAs Configurable Logic Block. [online]. Xilinx Inc., 2014-11-07 [cit. 2014-25-12].
   URL http://www.xilinx.com/support/documentation/user\_guides/ug474\_7Series\_CLB.pdf
- [2] Altera SDK for OpenCL Overview. [online]. Altera Corporation, [cit. 2015-05-23]. URL https://www.altera.com/products/design-software/ embedded-software-developers/opencl/overview.html
- [3] Altera SoC Development Board. [online]. RocketBoards.org, [cit. 2015-05-23]. URL http://www.rocketboards.org/foswiki/Documentation/ AlteraSoCDevelopmentBoard
- [4] AMBA 4 AXI4-Stream Protocol. [online]. ARM, 2010-03-03 [cit. 2015-29-12]. URL https://silver.arm.com/download/download.tm?pv=1074010
- [5] AMBA AXI and ACE Protocol Specification. [online]. ARM, 2011-10-28 [cit. 2015-29-12].
   URL https://silver.arm.com/download/download.tm?pv=1377613
- [6] AN 307: Altera Design Flow for Xilinx Users. [online]. Altera Corporation, [cit. 2015-05-18]. URL https://www.altera.com/content/dam/altera-www/global/en\_US/pdfs/literature/an/an307.pdf
- [7] Architecture Matters: Choosing the Right SoC FPGA for Your Application. [online]. Altera Corporation, 2013-11 [cit. 2015-01-02]. URL http://www.altera.com/literature/wp/ wp-01202-embedded-system-soc-design-considerations.pdf
- [8] Avalon Interface Specifications. [online]. Altera Corporation, 2014-06-30 [cit. 2014-29-12]. URL http://www.altera.com/literature/manual/mnl\_avalon\_spec.pdf
- [9] AXI DMA v7.1. [online]. Xilinx Inc., 2015-04-01 [cit. 2015-05-10]. URL http://www.xilinx.com/support/documentation/ip\_ documentation/axi\_dma/v7\_1/pg021\_axi\_dma.pdf
- [10] Comparing Altera APEX 20KE & Xilinx Virtex-E Logic Densities. [online]. Altera Corporation, [cit. 2015-01-09].

URL http://www.altera.com/products/devices/apex/features/ apx-compdensity.html

- [11] Cyclone V Hard Processor System Technical Reference Manual. [online]. Altera Corporation, 2014-12-15 [cit. 2014-25-12].
   URL http://www.altera.com/literature/hb/cyclone-v/cv\_5v4.pdf
- [12] Dokumentace JIC ovladače. [online]. Viktorin J., [cit. 2015-05-12]. URL https://ant-2.fit.vutbr.cz/projects/rsoc-driver/wiki/ Dokumentace\_JIC\_ovlada%C4%8De
- [13] EBV SoCrates Evaluation Board. [online]. RocketBoards.org, [cit. 2015-05-23]. URL http://www.rocketboards.org/foswiki/Documentation/ EBVSoCratesEvaluationBoard
- [14] Embedded Peripheral IP User Guide. [online]. Altera Corporation, [cit. 2015-05-10]. URL https: //www.altera.com/en\_US/pdfs/literature/ug/ug\_embedded\_ip.pdf
- [15] Logic Elements and Logic Array Blocks in Cyclone IV Devices. [online]. Altera Corporation, 2009-11 [cit. 2014-25-12]. URL http://www.altera.com/literature/hb/cyclone-iv/cyiv-51002.pdf
- [16] SDAccel Development Environment. [online]. Xilinx Inc., [cit. 2015-05-23]. URL http://www.xilinx.com/products/design-tools/sdx/sdaccel.html
- [17] SDSoC Development Environment. [online]. Xilinx Inc., [cit. 2015-05-23]. URL http://www.xilinx.com/products/design-tools/sdx/sdsoc.html
- [18] SDx Development Environments. [online]. Xilinx Inc., [cit. 2015-05-23]. URL http://www.xilinx.com/products/design-tools/sdx.html
- [19] Software Defined Specification Environment for Networking. [online]. Xilinx Inc., [cit. 2015-05-23]. URL http://www.xilinx.com/products/design-tools/sdx/sdaccel.html
- [20] Zynq-7000 All Programmable SoC Technical Reference Manual. [online]. Xilinx Inc., 2014-11-19 [cit. 2014-25-12]. URL http://www.xilinx.com/support/documentation/user\_guides/ ug585-Zynq-7000-TRM.pdf
- [21] Viktorin, J.: HW/SW Co-design for the Xilinx Zynq Platform, Master's thesis. FIT VUT v Brně, 2013.

## Appendix A

## Contents of the included CD

Since the *RSoC Framework* is a proprietary system source codes includes only components and drivers for integration on the *Altera* platform.

For the demonstration a pre-compiled binary of the system is included as well as encrypted *IP* core with license for a custom compilation.

The included CD contains following items:

- buildroot-external/ external buildroot package for building Linux for the *Altera Cyclone V SoC Development Board* with the multiplatform driver (as included on the SD Card image).
- ip/ contains the necessary *IP* directory (with sources of the top-level design and encrypted sources of the trial version of the *RSoC Bridge*).
- license/ contains the time unlimited license for encrypted *IP*.
- sdimage / contains the image of the SD Card to demonstrate the *RSoC Framework* on the *Altera Cyclone V SoC Development Board* with complete *FPGA* design and software (Linux with necessary drivers and software).
- project / contains the Quartus project archive for compiling the design targeted for the Altera Cyclone V SoC Development Board (as included on the SD Card image).
- src\_altera/ contains the *HDL* sources of components described in the Section 4.2.
- src\_driver/ contains the sources of the *Altera* low-level part of the multiplatform driver described in the Section 4.4.
- src\_thesis/ contains the LaTeX sources of this work.

### Appendix B

## How to build and use the RSoC Framework on the Altera Cyclone V SoC Development Board

**Prerequisites** Quartus II (minimal version 14.0) for compiling FPGA design, Linux installation for building embedded Linux with drivers, *Altera Cyclone V SoC Development Board* for running the demo.

For the correct function of the Quartus and buildroot copy the contents of the CD somewere on your disk.

#### B.1 Compiling the Quartus design for the Altera Cyclone V SoC Development Board

- 1. Open Quartus II.
- 2. Click Project/Restore Archived Project. Navigate to

project/socdev\_generator\_blackhole\_loopback\_demo.qar and restore archived project.

- 3. Now you have to open Qsys by clicking Tools/Qsys. In Qsys select to open "soc\_system.qsys" file in project's root directory.
- 4. When Qsys system is loaded click Tools/Options in the opened Options window click Add and navigate to ip/ directory and click open.
- 5. Now click Generate HDL and select VHDL files and click generate.
- 6. Click finish.
- 7. Now in back in Quartus open Tools/License setup and navigate to license license/ directory and select "license-rsoc.dat" and click open.
- 8. The Quartus project is now ready to compile. Click the Compile button to compile.
- 9. After a successful compilation click File/Convert Programming Files. There in the opened windows select Programming File Type to Raw Binary File (\*.rbf), down

on input files to convert click add and navigate to output\_files/ in the project's root and click on *"soc\_system.sof"*. Now click generate. This will generate \*.rbf which you can copy to SD Card (named soc\_system.rbf) and it will automatically load the design to *FPGA* during boot.

#### B.2 Compiling the Linux for the Altera Cyclone V SoC Development Board

- 1. Create a working directory e.g. "my-workdir".
- 2. Copy the buildroot-external folder to this directory.
- 3. Navigate to working directory:
  - \$ cd my-workdir
- 4. Clone the buildroot to the working directory:
  - \$ git clone https://github.com/RehiveTech/buildroot.git
- 5. Now execute:

```
$ make -C buildroot O=`pwd`/altera_socdev \
BR2_EXTERNAL=`pwd`/buildroot-external altera_socdevkit_defconfig
```

6. Navigate to altera\_socdev directory:

```
$ cd altera_socdev
```

- 7. Run menuconfig:
  - \$ make menuconfig
- 8. Be sure to check rsocdrv in the User-provided options sub-menu (do not check direct or mmap ctrl). Also open Target options sub-menu in which change target ABI from EABIhf to EABI. Then select Exit and save the configuration when prompted.
- 9. Compile complete system by running \$ make. This will also download any necessary files.
- 10. When building is complete, the images will be located in images/ folder. The files necessary to run the demo are "rootfs.cpio.uboot", "uImage" and "socfpga.dtb".

#### **B.3** Running the demo

Even if you have built your own images (the reason is that they does not contain preloader and u-boot bootloader), use the prebuild SD Card image located in sdimage/ Copy the raw SD Card image to at least 4GB Micro-SD Card using following commands:

```
$ cd /path/to/sdimage/
$ unzip altera_socdev.img.zip
$ dd if=altera_socdev.img of=/dev/sdX bs=4096
# where sdX is where your SD Card is located
```

If you want to use your own build, copy your images and design to the SD Card now. Now you can use the provided demo:

- 1. Insert Micro SD Card into your Altera Cyclone V SoC Development Board.
- 2. Check the DIP switches to be in the default positions.
- 3. Power on your board.
- 4. Connect the USB cable from your computer to UART on the Altera Cyclone V SoC Development Board.
- 5. Connect to the serial console:

\$ screen /dev/ttyUSB0 115200

6. Login with "root" and no password.

Welcome to Buildroot buildroot login: root

7. Probe the RSoC Driver.

# modprobe rsocdrv

8. Set up generator to generate data.

```
# rsocdrv-user /dev/rsoc-acc.0 0 0xffff
```

9. Read the data from the generator (the throughput will be shown when -m is used):

# rsocdrv-read /dev/rsoc-acc.0 -1 4096 -m > /dev/null
avg 148.802947 MB/s 1.569454 GB (10.800331 s)

10. Write the data to blackhole (again the throughput will be shown):

```
# rsocdrv-write /dev/rsoc-acc.1 -1 4096 -m < /dev/zero
avg 136.546968 MB/s 1.173477 GB (8.800200 s)</pre>
```

11. Send data through loopback (use read and write).

```
# rsocdrv-write /dev/rsoc-acc.2 < /dev/zero &
# rsocdrv-read /dev/rsoc-acc.2 -1 4096 -m > /dev/null
avg 33.232859 MB/s 624.808594 MB (18.800928 s)
```