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 | Bc. BRUNO KREMEL
AUTHOR

VEDOUCÍ PRÁCE | Ing. PAVOL KORČEK
SUPERVISOR

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 evaulates 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 Brně, 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.

Contents

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 *System-on-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.

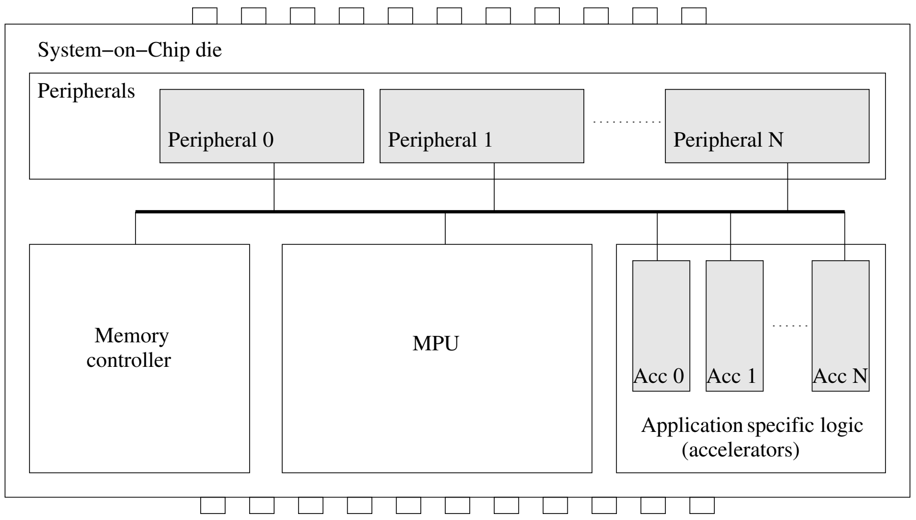


Figure 2.1: Typical System-on-Chip

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*.

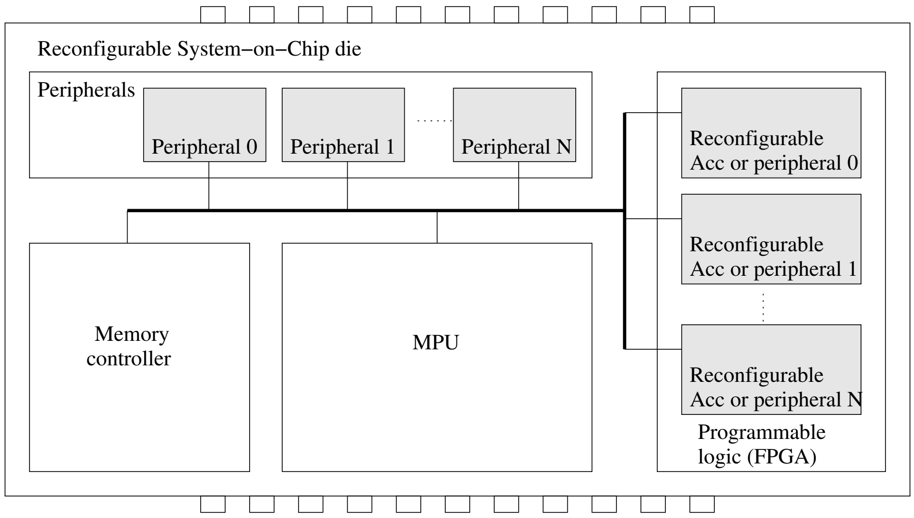

Reconfigurable System–on–Chip die
Peripherals
Peripheral 0
Peripheral 1
Peripheral N
Memory controller
MPU
Reconfigurable Acc or peripheral 0
Reconfigurable Acc or peripheral 1
Reconfigurable Acc or peripheral N
Programmable logic (FPGA)

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 *IP*s. 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.

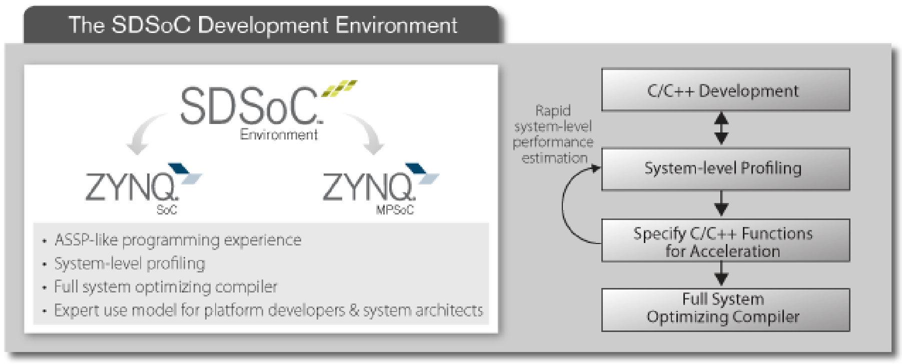

The SDSoC Development Environment
SDSoC Environment
ZYNQ SoC
ZYNQ MPSoC
• ASSP-like programming experience
• System-level profiling
• Full system optimizing compiler
• Expert use model for platform developers & system architects
Rapid system-level performance estimation
C/C++ Development
System-level Profiling
Specify C/C++ Functions for Acceleration
Full System Optimizing Compiler

Figure 2.3:

Figure 2.4: The development flow with Xilinx SDSoC [17]

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.

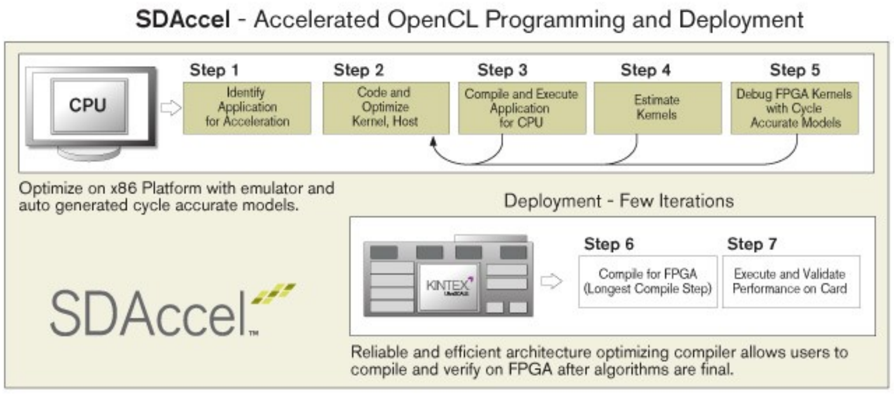

SDAccel - Accelerated OpenCL Programming and Deployment
CPU
Step 1
Identify Application for Acceleration
Step 2
Code and Optimize Kernel, Host
Step 3
Compile and Execute Application for CPU
Step 4
Estimate Kernels
Step 5
Debug FPGA Kernels with Cycle Accurate Models
Optimize on x86 Platform with emulator and auto generated cycle accurate models.
Deployment - Few Iterations
Step 6
Compile for FPGA (Longest Compile Step)
Step 7
Execute and Validate Performance on Card
KINTEX
SDAccel
Reliable and efficient architecture optimizing compiler allows users to compile and verify on FPGA after algorithms are final.

Figure 2.5:

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.

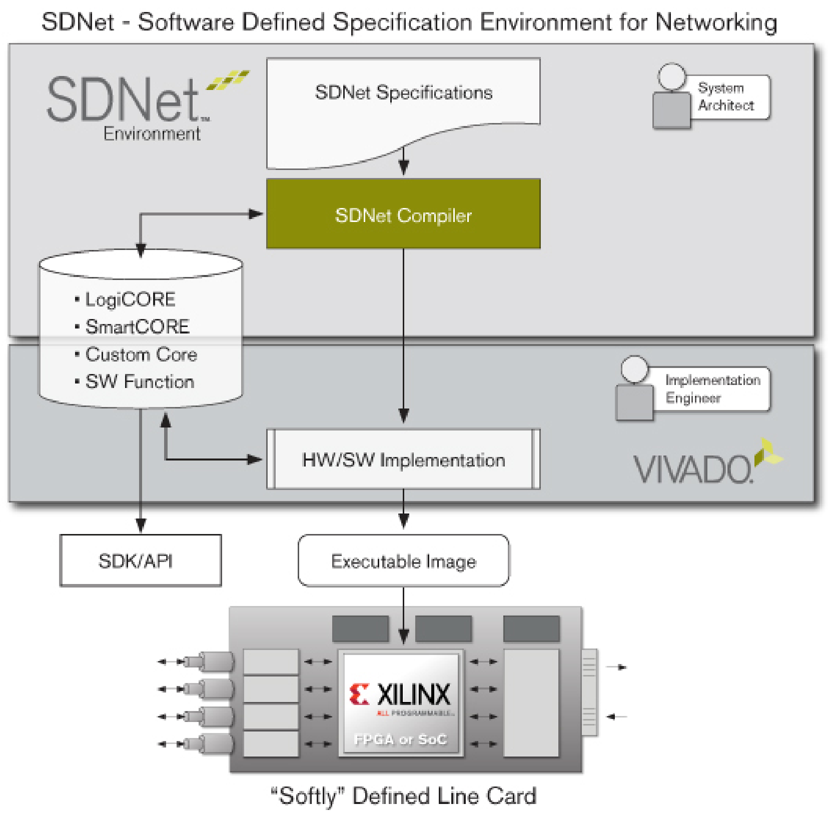


Figure 2.7:

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.

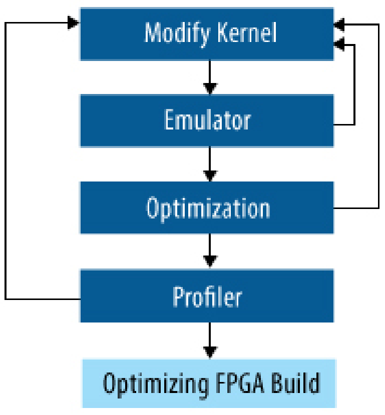


Figure 2.9:

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

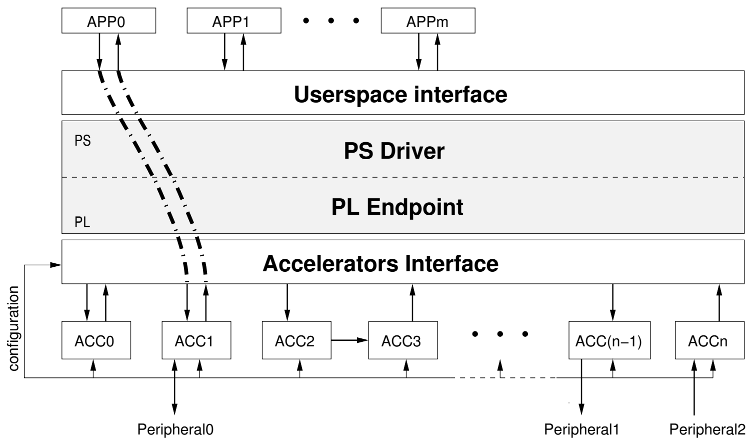


Figure 2.11: Basic design of RSoC Bridge

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).

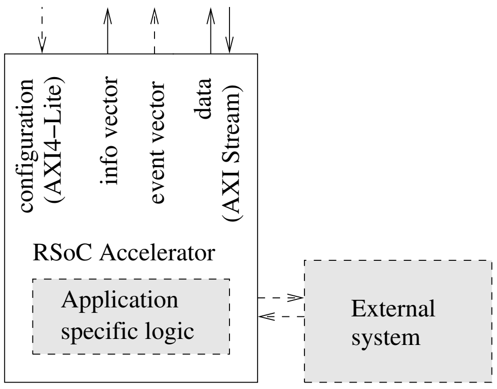


Figure 2.12: RSoC accelerator interface

- 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 Accelerator*s 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 *DMA* controller. 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 *ARM* uses this bus on their *SoC microprocessor* solutions.

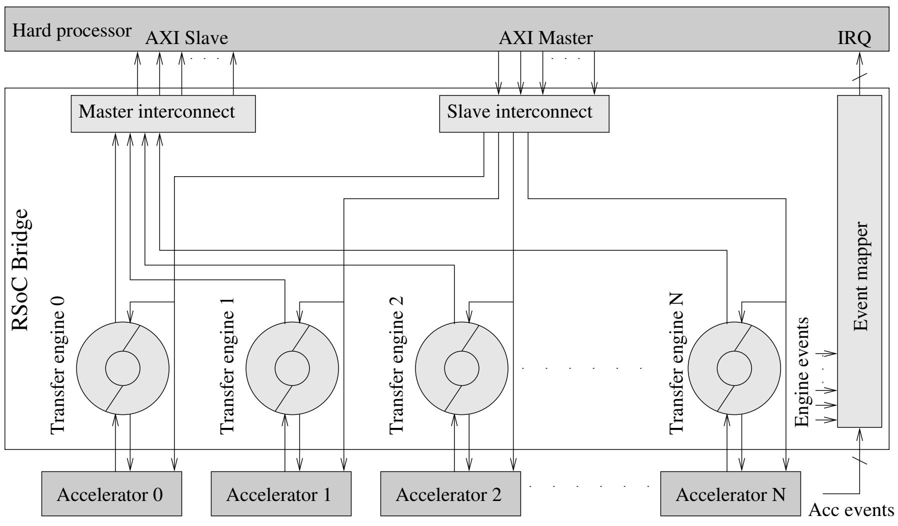


Figure 2.13: The basic RSoC Bridge architecture

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

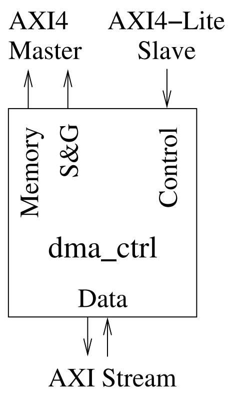


Figure 2.14: Interface for DMA controllers

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* 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.

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 K to 462 K LE	28 K to 444 K LC

Table 3.1: Parameters of the Altera and Xilinx SoCs

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
- 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

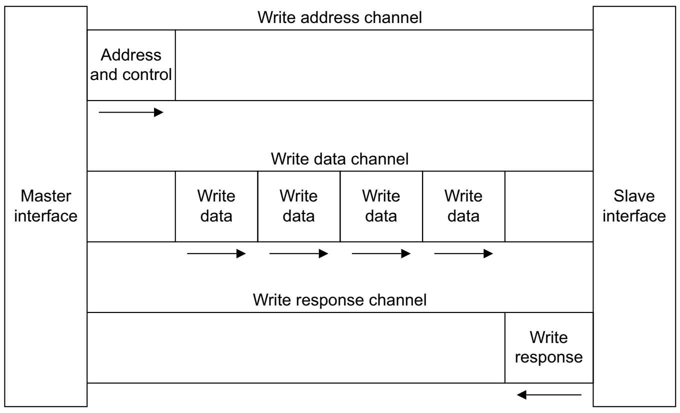


Figure 3.1: Basic architecture of the AXI write transaction [5]

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

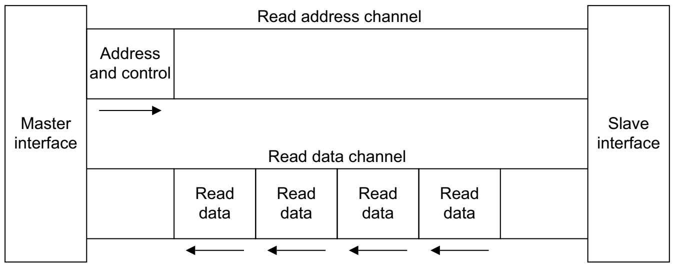


Figure 3.2: Basic architecture of the AXI read transaction [5]

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.

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

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

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 asserted for one cycle for each read access received.
WAITREQUEST	*Slave*	Asserted by the slave when it is unable to respond to a read/write request. 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

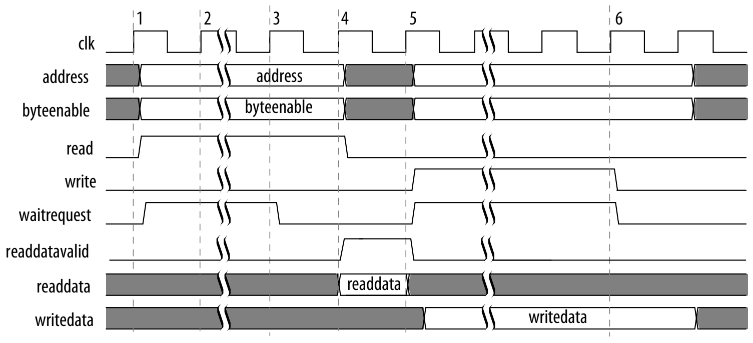


Figure 3.3: Read and write transfer with `waitrequest` [8]

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.

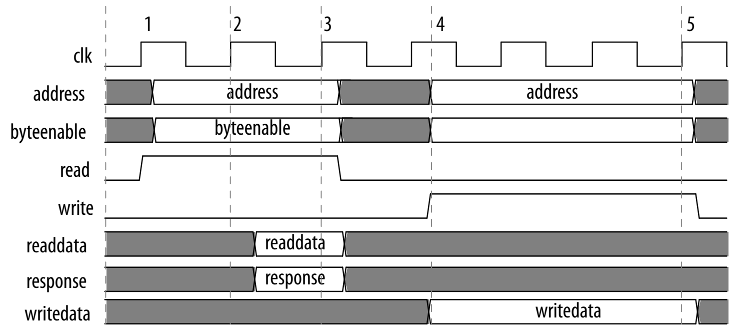


Figure 3.4: Read and write transfer with Fixed Wait-States at Slave interface [8]

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.

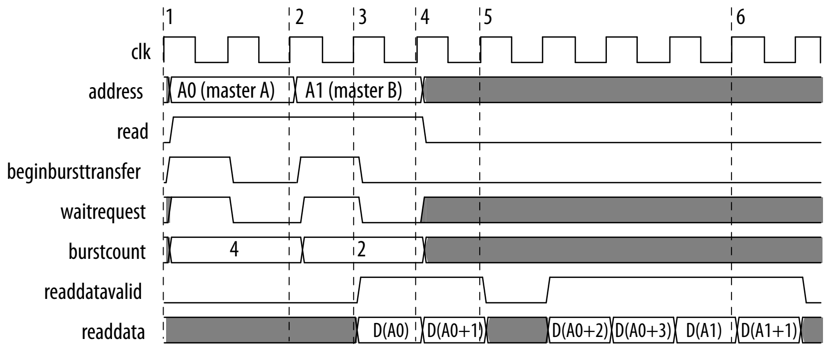


Figure 3.5: Bursting read transaction [8]

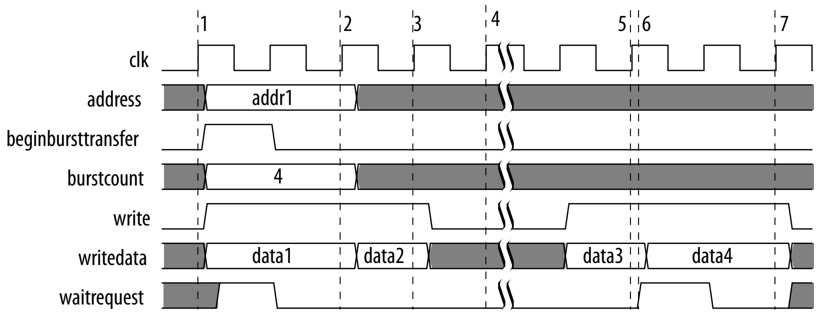


Figure 3.6: Bursting write transaction [8]

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 during 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

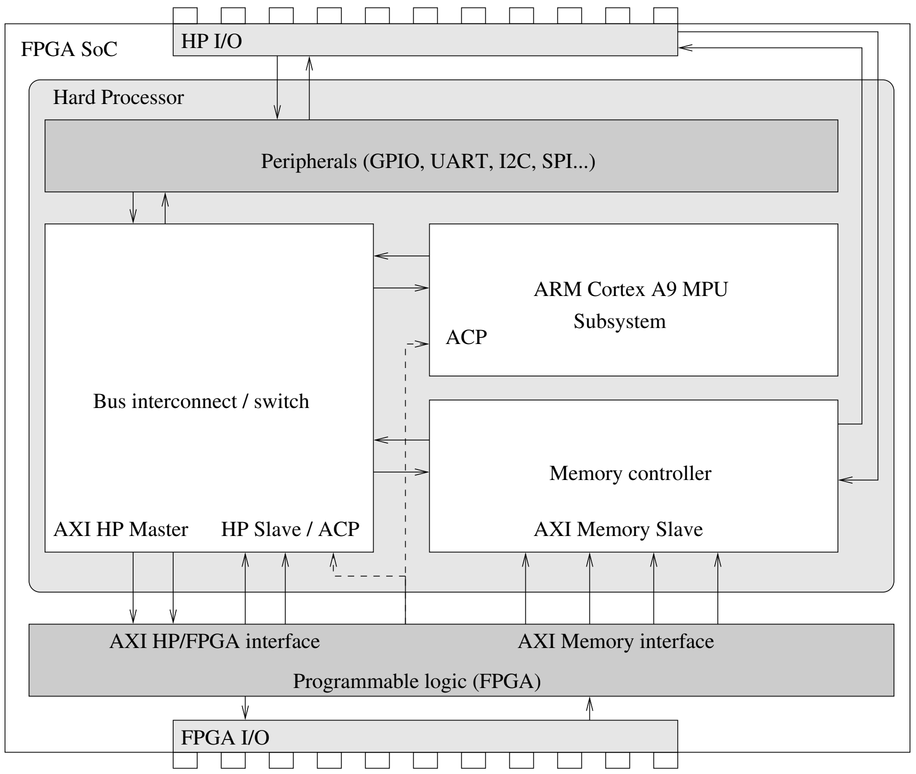


Figure 3.7: Basic architecture common to Xilinx and Altera FPGA SoCs

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	1x 32/64/128 bit AXI (CPU to FPGA) 1x 32/64/128 bit AXI (FPGA to CPU)	2x 32 bit AXI (CPU to FPGA) 2x 32 bit AXI (FPGA to CPU)
Low-Latency Processor/FPGA Interconnect	1x 32 bit AXI (CPU to FPGA)	not available (shared with high-bandwidth)
Processor/FPGA interconnect data width (theoretical max. bandwidth)	32/64/128 (10.8GB/s)	32 (4.8GB/s)

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 interface (FPGA to DRAM)	4x 64 bit AXI (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 read ports 4 write ports	4 x64 read ports 4 x64 write ports
Maximum interconnect to processor DDR Hard Memory Controller ports (connection type)	6 command/response ports 4 read ports 4 write ports (direct)	2 x64 read port 2 x64 write port (multiplexed)

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 (CPU to FPGA) (FPGA to CPU)	HPS-to-FPGA (H2F) FPGA-to-HPS (F2H)	General Purpose (AXI_GP)
Low-Latency Processor/FPGA Interconnect (CPU to FPGA)	HPS-to-FPGA Lightweight (H2F LW)	not available
ACP	FPGA-to-HPS	ACP (AXI_ACP)
FPGA-to-DDR Memory interconnect port	FPGA-to-HPS SDRAM (F2H SDRAM)	High Performance (AXI_HP)

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).

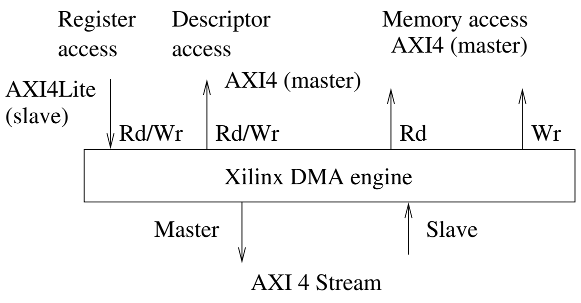


Figure 3.8: Xilinx DMA controller interfaces

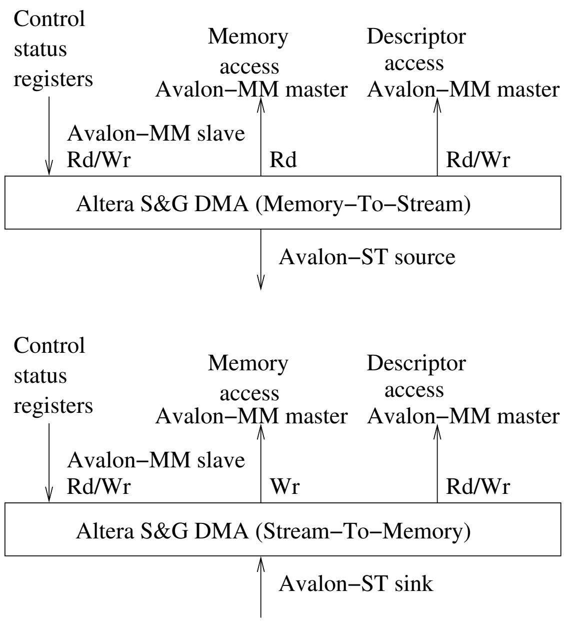

Control status registers
Memory access Avalon–MM master
Descriptor access Avalon–MM master
Avalon–MM slave Rd/Wr
Rd
Rd/Wr
Altera S&G DMA (Memory–To–Stream)
Avalon–ST source
Control status registers
Memory access Avalon–MM master
Descriptor access Avalon–MM master
Avalon–MM slave Rd/Wr
Wr
Rd/Wr
Altera S&G DMA (Stream–To–Memory)
Avalon–ST sink

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

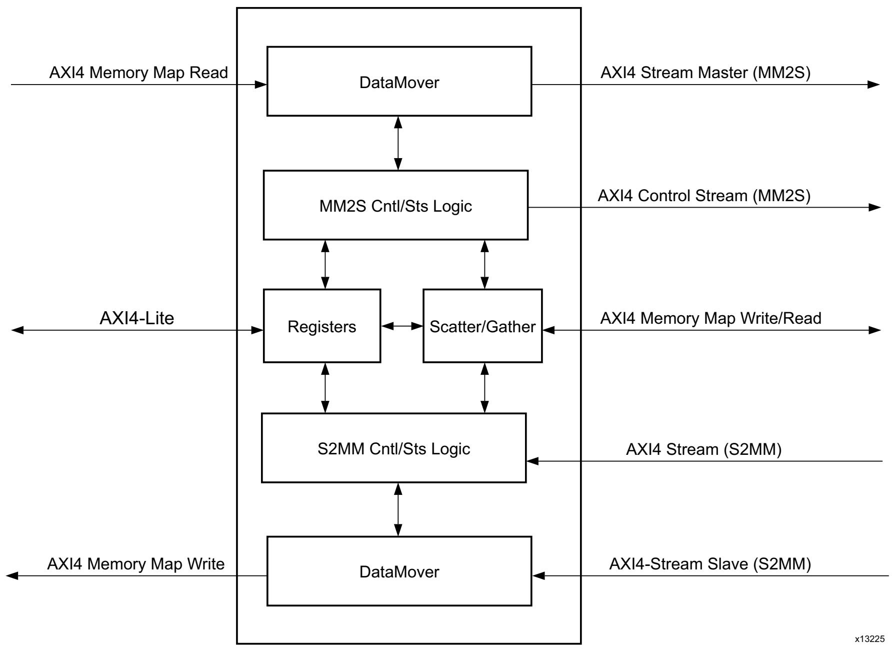


Figure 3.10: AXI DMA Block Diagram [9]

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].

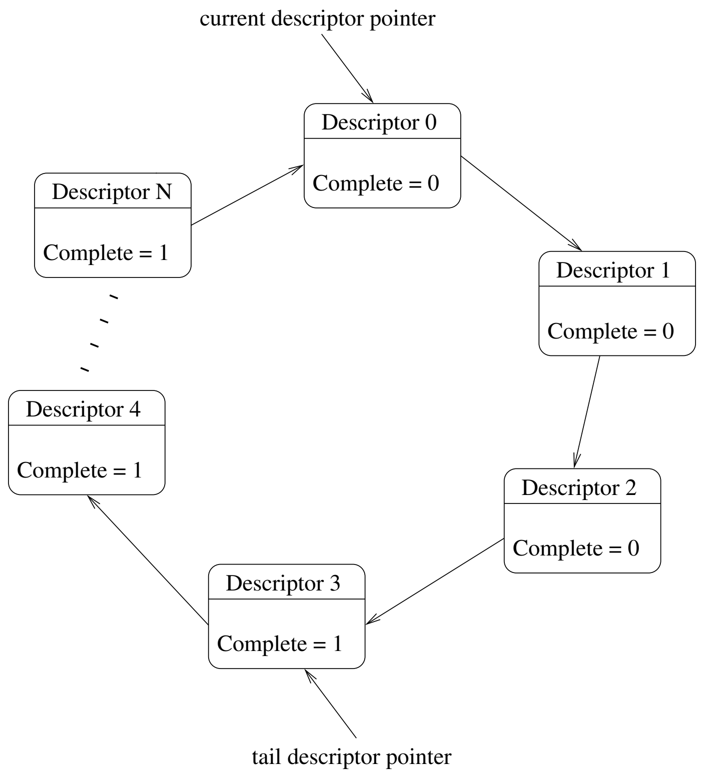


Figure 3.11: Xilinx DMA controller descriptor processing

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).

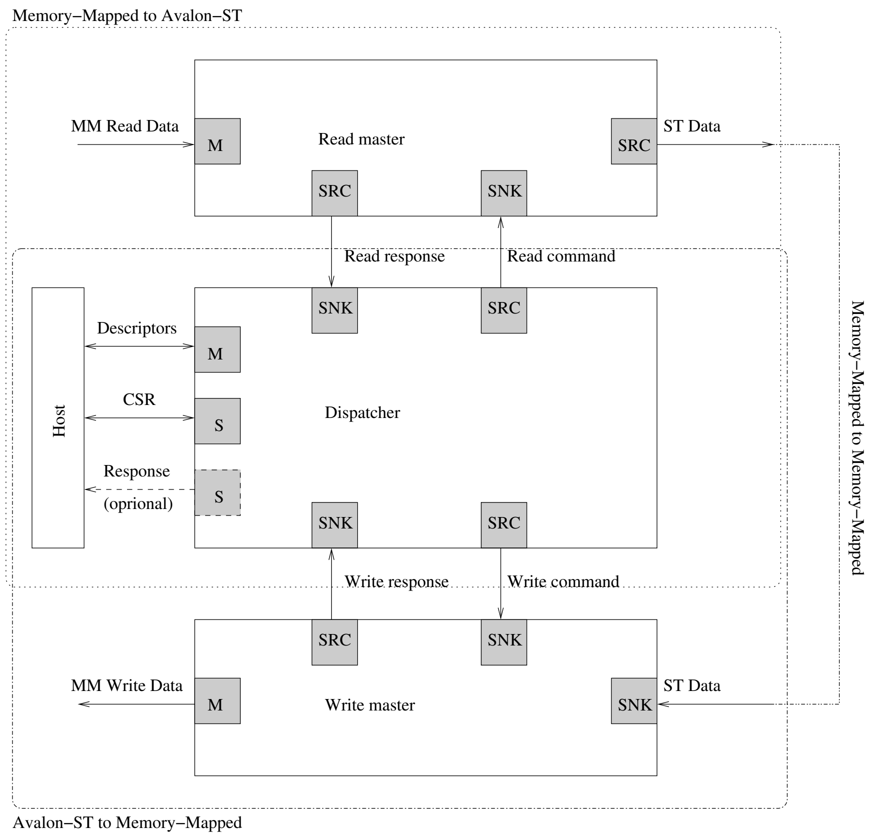


Figure 3.12: Altera Modular Scatter-Gather DMA Block Diagram [14]

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

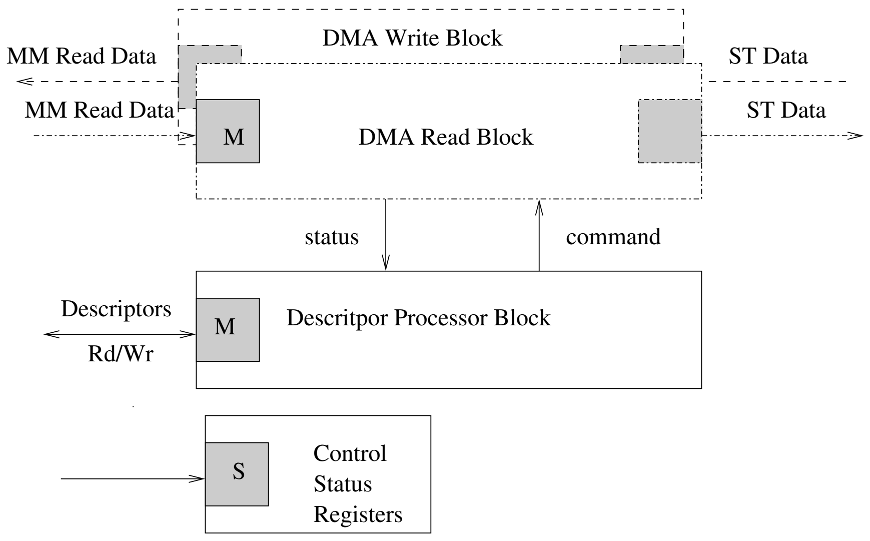


Figure 3.13: Altera Scatter-Gather DMA Block Diagram [14]

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.

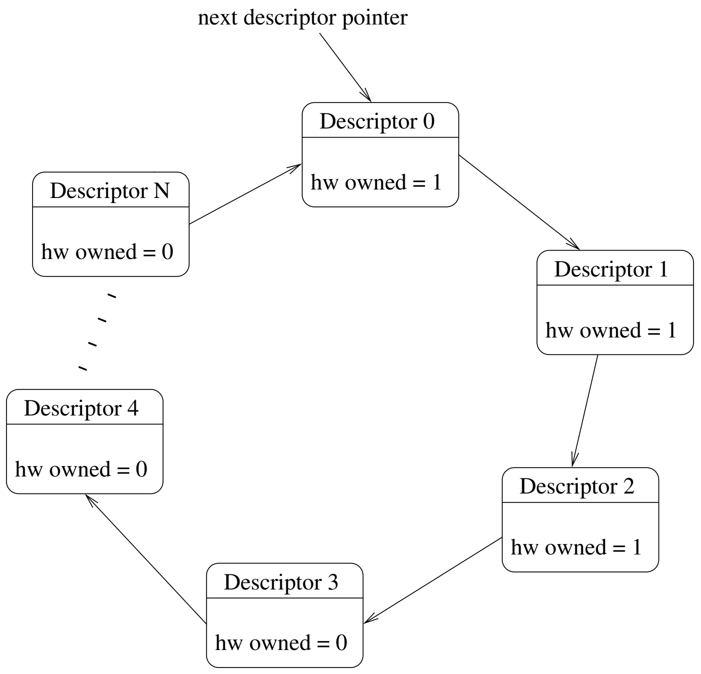


Figure 3.14: Altera DMA controller descriptor processing

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 design flow shown on the Figure 3.15.

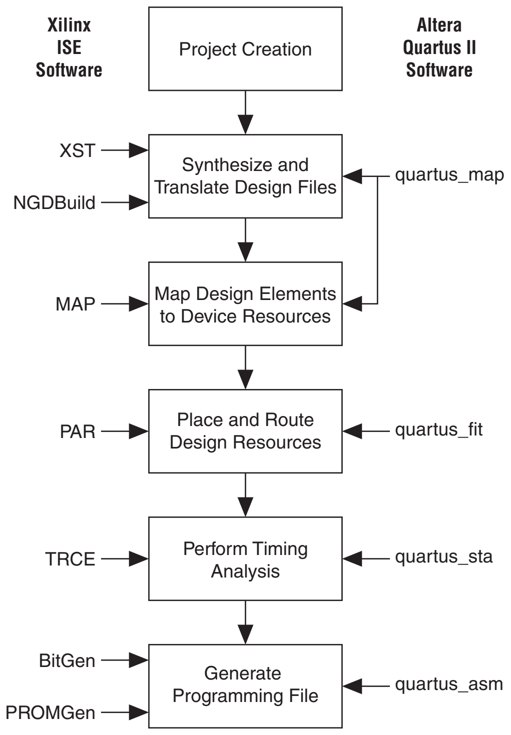

Xilinx ISE Software
Altera Quartus II Software
Project Creation
XST
NGDBuild
Synthesize and Translate Design Files
quartus_map
MAP
Map Design Elements to Device Resources
PAR
Place and Route Design Resources
quartus_fit
TRCE
Perform Timing Analysis
quartus_sta
BitGen
PROMGen
Generate Programming File
quartus_asm

Figure 3.15: Basic design flow

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

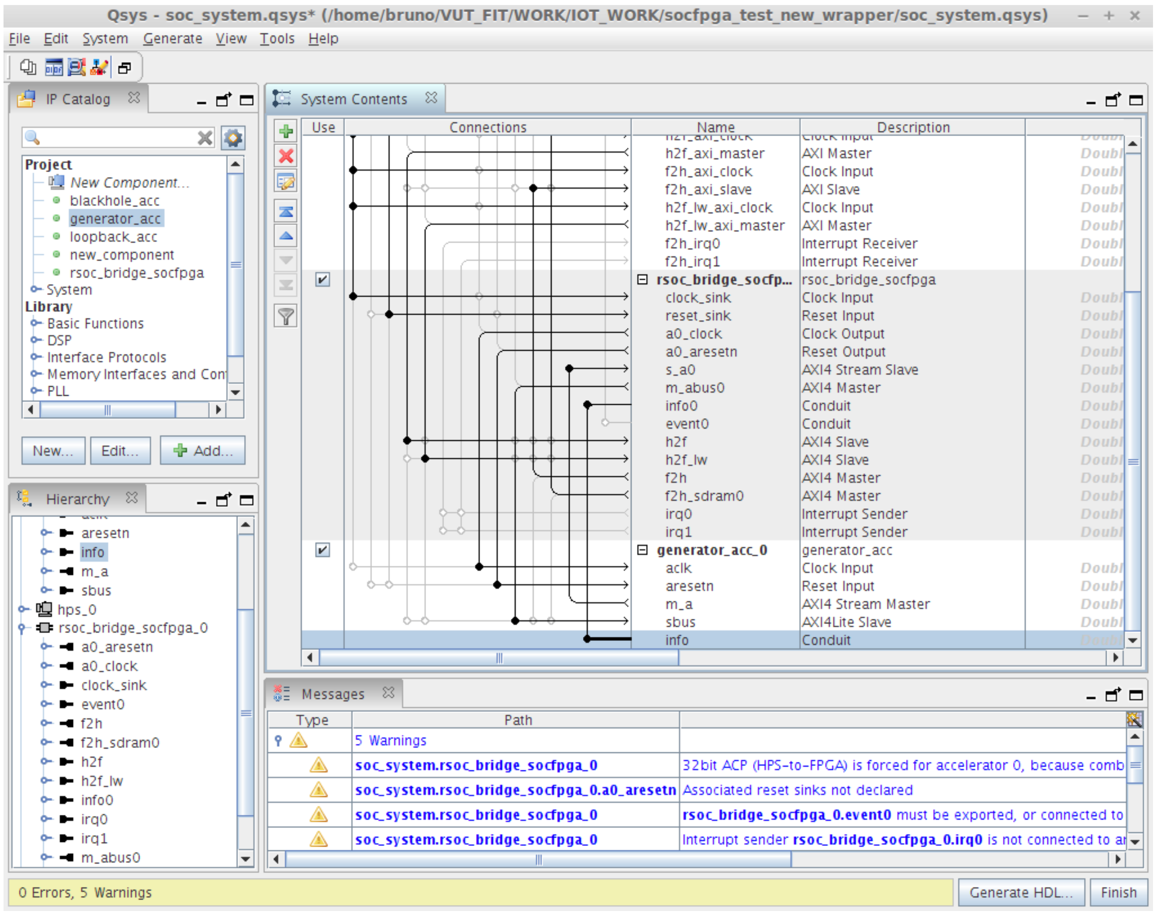
Both vendors provide a system integration tool, that is graphical interface tool in which user can configure IPs and put together a system from vendor's and third-party IPs. To provide user a consistent and familiar environment, the *RSoC framework* must support these tools.

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*, *AXI4 Lite* and *AXI Stream*).



Name	Description
h2f_axi_master	AXI Master
f2h_axi_clock	Clock Input
f2h_axi_slave	AXI Slave
h2f_lw_axi_clock	Clock Input
h2f_lw_axi_master	AXI Master
f2h_irq0	Interrupt Receiver
f2h_irq1	Interrupt Receiver
rsoc_bridge_socfp...	rsoc_bridge_socfpga
clock_sink	Clock Input
reset_sink	Reset Input
a0_clock	Clock Output
a0_aresetn	Reset Output
s_a0	AXI4 Stream Slave
m_abus0	AXI4 Master
info0	Conduit
event0	Conduit
h2f	AXI4 Slave
h2f_lw	AXI4 Slave
f2h	AXI4 Master
f2h_sdram0	AXI4 Master
irq0	Interrupt Sender
irq1	Interrupt Sender
generator_acc_0	generator_acc
aclk	Clock Input
aresetn	Reset Input
m_a	AXI4 Stream Master
sbus	AXI4Lite Slave
info	Conduit

Type	Path	
⚠	5 Warnings	
⚠	soc_system.rsoc_bridge_socfpga_0	32bit ACP (HPS-to-FPGA) is forced for accelerator 0, because comb
⚠	soc_system.rsoc_bridge_socfpga_0.a0_aresetn	Associated reset sinks not declared
⚠	soc_system.rsoc_bridge_socfpga_0	rsoc_bridge_socfpga_0.event0 must be exported, or connected to
⚠	soc_system.rsoc_bridge_socfpga_0	Interrupt sender rsoc_bridge_socfpga_0.irq0 is not connected to ar

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

This chapter is not available in the public version.

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 Development 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].

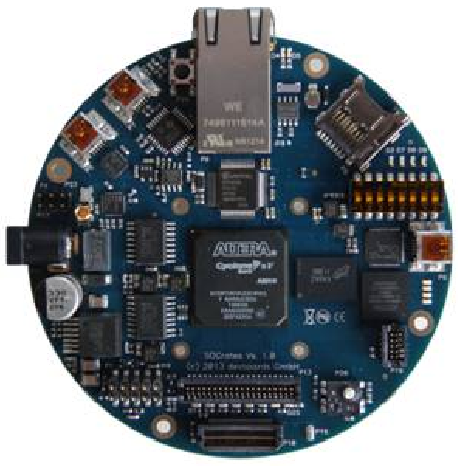


Figure 5.1: EBV Socrates Starter Kit [13]

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].

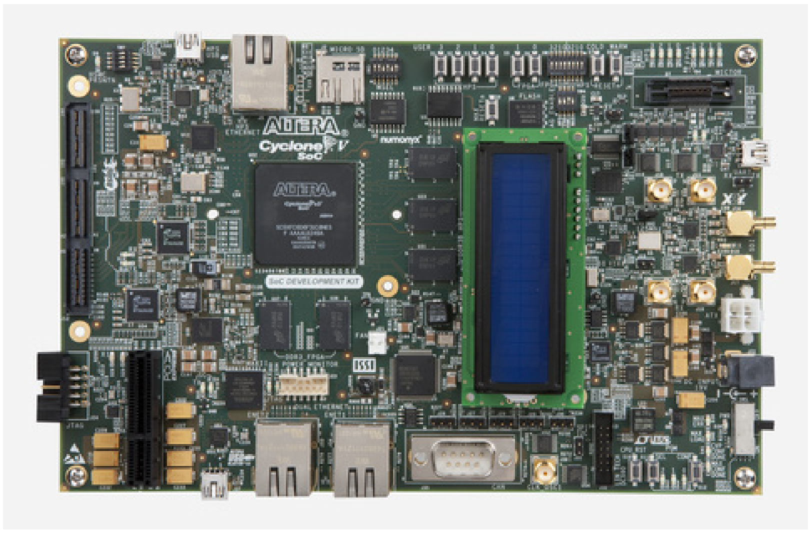


Figure 5.2: Altera Cyclone V SOC Development Board [3]

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

Measurement	explicit sync of caches	dma_alloc_coherent
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 MB/s
16 kB		
Read and ack descriptors	225 MB/s	225 MB/s
Do vmsplice to /dev/null	168 MB/s	225 MB/s
Access every data word	137 MB/s	30 MB/s
32 kB		
Read and ack descriptors	225 MB/s	226 MB/s
Do vmsplice to /dev/null	166 MB/s	226 MB/s
Access every data word	140 MB/s	30 /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 MB/s	276 MB/s
Do vmsplice to /dev/null	145 MB/s	276 MB/s
Access every data word	119 MB/s	30 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 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 MB/s	364 MB/s
Do vmsplice to /dev/null	176 MB/s	364 MB/s
Access every data word	132 MB/s	30 /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 buffer size	RX	TX	Loopback
4k	149 MB/s	138 MB/s	34 MB/s
8k	165 MB/s	120 MB/s	102 MB/s
16k	174 MB/s	120 MB/s	114 MB/s
32k	177 MB/s	147 MB/s	109 MB/s

Table 5.3: Performance of the multiplatform driver on Altera

Kernel buffer size	RX	TX	Loopback
4k	146 MB/s	173 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 **??**.
- `src_driver/` — contains the sources of the *Altera* low-level part of the multiplatform driver described in the Section 3.5.
- `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)
    ```

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)
    ```