VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

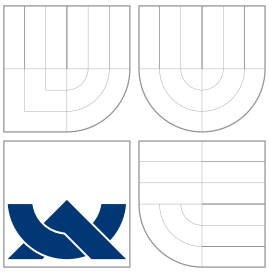
HW/SW CO-DESIGN FOR THE XILINX ZYNQ PLATFORM

DIPLOMOVÁ PRÁCE
MASTER THESIS

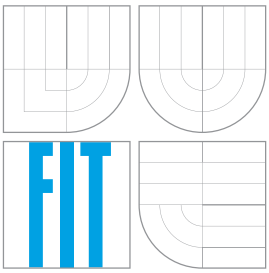
AUTOR PRÁCE
AUTHOR

Bc. JAN VIKTORIN

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HW/SW CO-DESIGN NA PLATFORMĚ XILINX ZYNQ

HW/SW CO-DESIGN FOR THE XILINX ZYNQ PLATFORM

DIPLOMOVÁ PRÁCE
MASTER THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN VIKTORIN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PAVOL KORČEK

BRNO 2013

Abstrakt

Tato práce se zabývá možnostmi pro HW/SW codesign na platformě Xilinx Zynq. Na základě studia rozhraní mezi částmi Processing System (ARM Cortex-A9 MPCore) a Programmable Logic (FPGA) je navržen abstraktní a univerzální přístup k vývoji aplikací, které jsou akcelerovány v programovatelném hardwaru na tomto čipu a běží nad operačním systémem Linux. V praktické části je pro tyto účely navržen framework určený pro Zynq, ale také pro jiné obdobné platformy. Žádný takový framework není v současné době k dispozici.

Abstract

This work describes a novel approach of HW/SW codesign on the Xilinx Zynq and similar platforms. It deals with interconnections between the Processing System (ARM Cortex-A9 MPCore) and the Programmable Logic (FPGA) to find an abstract and universal way to develop applications that are partially offloaded into the programmable hardware and that run in the Linux operating system. For that purpose a framework for HW/SW codesign on the Zynq and similar platforms is designed. No such framework is currently available.

Klíčová slova

Zynq, Linux, FPGA, AXI, SoC, HW/SW Codesign

Keywords

Zynq, Linux, FPGA, AXI, SoC, HW/SW Codesign

Citation

Jan Viktorin, HW/SW Co-design for the Xilinx Zynq Platform, diplomová práce, Brno. FIT VUT v Brně, 2013.

HW/SW Co-design for the Xilinx Zynq Platform

Declaration

I confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the document.

Jan Viktorin
Brno, 22. 5. 2013

Acknowledgment

I would like to thank Pavol Korček for reviewing my thesis. I am happy to have such a supportive supervisor. This work was supported by the project *Modern Tools for Detection and Mitigation of Cyber Criminality on the New Generation Internet.*

Contents

Figures

Glossary

ASIC. Application-Specific Integrated Circuit. Implements an application specific function.

ASSP. Application-Specific Standard Product. Implements a specific function for wide market.

CPIO. File format used to archive files.

DMA. Direct Memory Access enables a hardware unit to access the memory without CPU intervention.

DSP. Digital Signal Processor is a processor optimized for digital signal processing.

FPGA. Field-Programmable Gate Array is an integrated circuit configured by a customer designer after manufacturing.

GPU. Graphics Processing Unit. It is a specialized electronic circuit designed to accelerate graphical computations.

GZIP. Data compressor.

HDL. Hardware Description Language. Examples are: Verilog, VHDL.

HMAC. Keyed-hash Message Authentication Code. It is used to verify the data integrity and authentication of a message.

IP core. Intelectual Property core is a reusable unit of logic.

ISA. Industry Standard Architecture is a computer bus standard for IBM PC.

JTAG. Joint Test Action Group is a standard used for testing PCBs.

MMC, NAND, NOR. Multi Media Card is a flash memory card standard based on NAND and NOR technologies.

MPCore. Multicore processing system.

PDU. Protocol Data Unit.

Quad-SPI. Serial Peripheral Bus extended to use four data wires.

SD, SDIO. Secure Digital is a non-volatile memory card format. SDIO extends SD to cover I/O functions.

SMC. Static Memory Controller.

VHDL. VHSIC Hardware Description Language.

1 Introduction

The market of computing systems is led by the embedded systems. The embedded systems drive the mobile phones and smartphones, tablets, little USB sticks, set-top boxes, network routers, washing machines, thermostats, cars, planes and many other machines. People usually do not consider them to be computing systems, they do not think about them. What is important (and maybe interesting at the same time), there are usually very different requirements among such systems.

Some systems are time-critical and must be fault tolerant, others can fail occasionally and are allowed to respond with some latency. Just compare a unit that controls the amount of fuel that is coming to the engine of a car and a TV set-top box. The car unit must never fail or provide a fast recovering from a failure. There must not be delays between using the car accelerator and the increase of fuel coming to the engine. A set-top box can fail, can delay some operations, but it must be able to decode high-quality video real-time.

In the context of embedded systems there are many application specific devices. Each of them are optimized either for speed, low power consumption, high throughput of data and others. These devices are usually ASIC chips—hardwired in the sillicon and providing only a small amount of functions. But it is very common to integrate some of them into a single component that is called System-on-Chip (SoC). A SoC consists of a processor and some peripherals, and accelerators integrated tightly together using a bus system. This reduces the power consumption, the price of the chip and the shorter wires provides faster connections.

This work considers a new possible approach to designing SoC. It is an integration of a processor and an FPGA on a single chip. The processor provides the architecture that executes an application or an operating system (OS). The FPGA logic allows developers to accelerate the software part, to bring a non-standard or previously not supported interface to the SoC, fix bugs in the hardware part and other possibilities. In fact it is not a new idea and it has already been implemented, but today a new generation of such devices is coming to the market. The first one is the Xilinx Zynq, and the others, for example Altera Cyclone V, are coming soon.

2 Designing embedded systems

An embedded system is conventionally defined as a piece of computer hardware running software designed to perform a specific task. Examples of such systems might be TV set-top boxes, smartcards, routers, etc. However, the distinction between an embedded system and a general purpose (personal) computer is becoming increasingly blurred. The mobile phones might perform just the basic task, making phone calls, but modern smartphones can run a complex operating system and a rich set of applications installed by the users. [12, p. 1-4]

2.1 Operating systems in embedded systems

Embedded systems can contain very simple 8-bit microcontrollers or some of the more complex 32 or 64-bit processors, such as the ARM family. [12, p. 1-4] Applications built on the more powerful processors can benefit from running an operating system (OS). An operating system serves two main purposes in the area of embedded systems:

1. it provides an abstraction layer for software to be less dependent on hardware, which makes the development of applications that sit on top of the OS easier, and
2. manages the various system hardware and software resources to ensure the entire system operates efficiently and reliably. [3, p. 383]

Every OS consists at least of a *kernel*. The kernel contains the main functionality of the OS, specifically the following features:

- process management,
- memory management, and
- input/output system management.

2.2 System-on-Chip architectures

A System-on-Chip (SoC) is an integrated circuit that implements most or all of the functions of a complete electronic system. The most fundamental characteristic of a SoC is its complexity. However, a memory chip, for example, may have a large amount of transistors, but its regular structure makes it a component and not a system. Many SoCs contain analog and mixed-signal circuitry for input/output (I/O). Although some high-performance I/O applications require a separate analog interface chip that serves as a companion to a digital SoC. The system may contain limited amount of memory, one or more instruction-set processors, specialized logic, busses, and other digital functions. The architecture of the system is generally tailored to the application rather than being a general-purpose chip. [6, p. 2] Systems will almost always have additional peripherals—typically including UARTs, interrupt controllers, timers, GPIO controllers, but also potentially quite complex blocks such as DSPs, GPUs, or DMA controllers. [12, p. 1-4]

Many product categories (cell phones, telecommunications and networking, digital televisions, etc.) do not use general-purpose computer architectures because such machines are not cost-effective or because they would not provide the necessary performance. At the high-end, general-purpose machines cannot keep up with the data rates for high-end video and networking. They also have difficulty providing reliable real-time performance. [6, p. 2]

The performance of the SoC design heavily depends upon the efficency of its bus structure. IP cores, the components of SoCs, are designed with many different interfaces and communication protocols. Integration of such cores in a SoC often requires insertion of suboptimal glue logic. To avoid this problem, standard on-chip bus structures were developed. There are several public specifications of bus architectures from leading manufacturers, e. g. CoreConnect from IBM, ABMA from ARM, Wishbone from Silicore Corporation, and others. [18]

2.3 AMBA AXI architecture

The AMBA AXI [8] is a point-to-point protocol (or better to say a family of protocols) developed by ARM suitable for high-bandwidth and low-latency SoC designs. Recently, it has been adopted by FPGA manufacturers such as Xilinx [22, p. 5] and Altera [7] as a common bus system for their soft IP cores. It also integrates better with ARM processors.

The AXI protocol is burst-based with five independent transaction channels: *read address*, *read data*, *write address*, *write data* and *write response*. The figures 2.1 and 2.2 show how the read and write transactions use the channels.

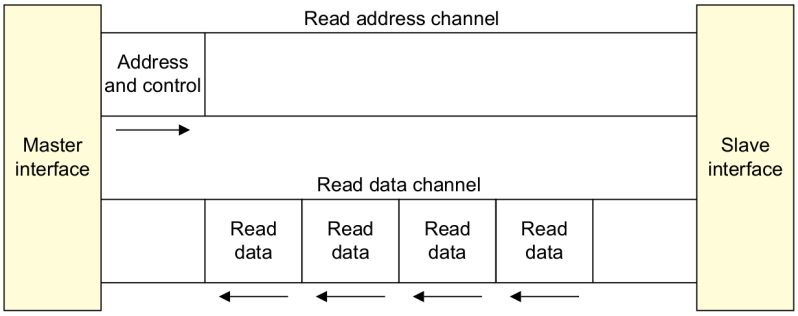


Figure 2.1: The read architecture of AXI. [8, p. 22]

A *master interface* initiates a transaction by specifying a source/target address of the transaction. Simultaneously the master specifies the size of the transaction, information about caching, privileges, QoS, or atomicity properties. There are optional user signals available.

After the transaction is initiated, another phase occurs. If it is a read transaction the slave now starts to send data to the master. In case of a write transaction the master starts to send data to the slave. When the master finishes, the slave returns a response that

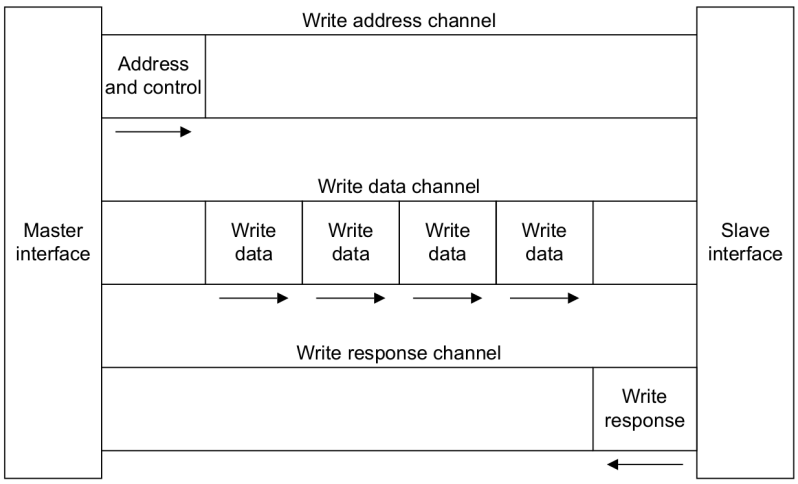


Figure 2.2: The write architecture of AXI. [8, p. 22]

allows the master to learn whether the write transaction succeeded or failed. Note that each phase uses a different independent physical channel. Each channel uses handshake signals `TVALID` and `TREADY`.

Such a design enables to use pipelining because there is no fixed relationship between the channels. This makes possible to trade-off between cycles of latency and maximum frequency of operation.

2.3.1 AXI4 protocol

The AXI4 is a follower of the AXI3 protocol. The differences are described later in the section *2.3.2 Differences between AXI3 and AXI4*. It is desirable to describe it in more detail because the AXI4 protocol is going to be widely used in the FPGA computing.

As it was mentioned there are five groups of signals: *read address*, *read data*, *write address*, *write data* and *write response*. There is another group the *global signals* which consists of `ACLK` and `ARESETn` signals. All signals are sampled on the rising edge of the `ACLK` signal. The `ARESETn` is a global reset signal with active low level.

The *read address* and *write address* are different only in names. The table 2.1 describes all signals briefly. Substitute the letter x for R when considering the *read address* signals and for W when considering the *write address* signals. The unspecified widths are configurable (useful especially for FPGAs). All signals have a prefix that determines the channel they belong to (e. g. A means *address*, AW means *write address*, B means *write response*).

All of the five interfaces use the handshake process based on the `VALID` and `READY` signals. The handshake process is described in detail in [8, p. 37–43]. The data (address or control) are transferred from source to destination when both `VALID` (by source) and `READY` (by destination) are asserted and the rising edge of `ACLK` occurs. There is an important restriction to avoid loops, the source must never wait until the `READY` signal is asserted. Instead

Read/write address			*Write data/Write response*			*Read data*		
Name	**Source**	**Width**	**Name**	**Source**	**Width**	**Name**	**Source**	**Width**
AxID	*Master*	-	WID	*Master*	-	RID	*Slave*	-
AxADDR	*Master*	-	WDATA	*Master*	-	RDATA	*Slave*	-
AxLEN	*Master*	8	WSTRB	*Master*	-	RRESP	*Slave*	2
AxSIZE	*Master*	3	WLAST	*Master*	1	RLAST	*Slave*	1
AxBURST	*Master*	2	WUSER	*Master*	-	RUSER	*Slave*	-
AxLOCK	*Master*	2	WVALID	*Master*	1	RVALID	*Slave*	1
AxCACHE	*Master*	4	WREADY	*Slave*	1	RREADY	*Master*	1
AxPROT	*Master*	3						
AxQOS	*Master*	4	BID	*Slave*	-			
AxREGION	*Master*	4	BRESP	*Slave*	2			
AxUSER	*Master*	-	BUSER	*Slave*	-			
AxVALID	*Master*	1	BVALID	*Slave*	1			
AxREADY	*Slave*	1	BREADY	*Master*	1			

Table 2.1: Signals of the five signal interfaces of the AXI4 protocol

whenever the source has data available it must assert the VALID signal independently on the destination. Once the VALID is asserted by the source it must remain asserted until the data transfer occurs.

Both data channels pass data in longer sequences—bursts. The last data transfer in the burst is marked by the appropriate LAST signal asserted. Each burst transaction is configurable during the address phase. The AxLEN signal specifies number of beats the transaction has (the limit is 256) and the AxSIZE defines the number of bytes transferred in one beat. A burst of only 1 data beat is possible. Note that the data stream must be of the length specified by the AxLEN and AxSIZE signals, see the restriction in [8, p. 44]. The AxBURST specifies how the slave modifies the address during the burst. The commonly implemented values are *FIXED* (the address is the some for every beat, useful for accessing a FIFO) and *INCR* (the address is incremented for each beat to access a sequential memory). The write data channel can provide STRB signal. It represents a bitmask that marks which bytes of the corresponding DATA signal are valid (it is a kind of byte-enable signal). [8, p. 49]

The Xilinx defines its own supported subset of AXI that is available to the Xilinx FPGA IPs. The subset is described in [22].

2.3.2 Differences between AXI3 and AXI4

There are currently two versions of the AXI protocol: AXI3 and AXI4. There are some minor differences between them. The AXI4 protocol changes the AXI3 protocol in the following ways, it

- defines an additional AXI4 slave write response dependency, [8, p. 42–43]
- extends the signal AxLEN from 4 bits to 8 bits, [8, p. 44]

- modifies the semantics of the bit `AxCACHE[1]` and this modification affects the bits `AxCACHE[3:2]` and memory types definitions as well, [8, p. 59–70, 72–73]
- removes support for write interleaving with different AWID values, [8, p. 79–81]
- has different transaction ordering model, [8, p. 83–88]
- intruduces a concept of single-copy atomicity, [8, p. 90–96]
- removes the support of locking transactions, [8, p. 96]
- adds QoS (Quality of Service) signaling, [8, p. 98]
- adds multiple regions to provide multiple logical interfaces, [8, p. 99]
- adds user-defined signals. [8, p. 100]

The AXI3 protocol is currently used on various chips (e. g. Xilinx Zynq). The newer AXI4 protocol is designed with backward compatibility in mind. The AXI3 and AXI4 devices can cooperate without changes if they satisfy few conditions. See the [8, p. 28–34] for details.

2.3.3 AXI4-Lite protocol

The AXI4-Lite [8, p. 122–127] is a limited version of the AXI4 protocol. It is intended for simplier interfaces with the register-style access. Such a communication does not need the full functionality of AXI and so it is possible to save resources on the chip for other logic.

Every AXI4-Lite burst transaction is of size 1 while using the full width of the data bus with no support of the extended attributes for caching, buffering and atomicity. The protocol is designed for interoperability with the standard AXI4 protocol without modifications. There is just one case where a modification is required—the connection AXI4 to AXI4-Lite, because AXI4-Lite does not support out-of-order transactions.

2.3.4 AXI4-Stream protocol

The AXI4-Stream protocol is a simplex—one way—bus (a link) from a master to a slave. There is no way for the slave to respond. It can stop the data flow just by the handshake signals. In fact, a subset of AXI4-Stream is used in the write and read data channels of the AXI4 protocol. The details are explained in [9]. Note that Xilinx defines its own subset of the AXI4-Stream protocol in [22].

All signals except of `TVALID` and `TREADY` are optional. There are predefined default values of the signals when any signal is missing. The list of signals used by the AXI4-Stream protocol can be find in the table 2.2. The n represents a number of bytes per data transfer.

The signal `TKEEP` represents a mask of valid bytes in the `TDATA` signal. The zero bits of the signal marks bytes that can be removed from the stream. It is possible to perform a transfer where the `TKEEP` signal contains only zeros (unless there is the `TLAST` signal asserted). The IP cores are not required to be able to process the zero bytes. [9, p. 8]

Name	Source	Width
TVALID	*Master*	1
TREADY	*Slave*	1
TDATA	*Master*	$8n$
TSTRB	*Master*	n
TKEEP	*Master*	n
TLAST	*Master*	1
TID	*Master*	-
TDEST	*Master*	-
TUSER	*Master*	-

Table 2.2: Signals of the AXI4-Stream protocol

The signal TSTRB is a mask that describes whether the associated byte is a data byte (one) or a position byte (zero). A data byte is a normal valid data byte. A position byte indicates a relative position of data bytes within the stream. The data asociated with position bytes is not valid. [9, p. 8]

The pairs of values of TKEEP and TSTRB have associated semantics:

- $\texttt{TKEEP}(i) = 1 \wedge \texttt{TSTRB}(i) = 1$: the i-th byte is valid and must be transmitted.
- $\texttt{TKEEP}(i) = 1 \wedge \texttt{TSTRB}(i) = 0$: the i-th byte indicates relative position.
- $\texttt{TKEEP}(i) = 0 \wedge \texttt{TSTRB}(i) = 0$: the i-th byte can be removed from the stream.
- $\texttt{TKEEP}(i) = 0 \wedge \texttt{TSTRB}(i) = 1$: represents a forbidden combination. [9, p. 9]

It is desirable to group bytes into structures called *packets* for more efficient processing. A packet is a similar concept to an AXI4 *burst*. The signal TLAST can be used by the destination to indicate a packet boundary. The protocol does not provide any explicit signaling of the start of a packet.

The signals TID and TDEST provide an identification of a packet transmitted over the stream. This is useful when a unit supports packet interleaving during the transfer. Any processing stage of an AXI-Stream can modify those values. The TID identifies the source of a packet on the link. The signal TDEST provides coarse routing information for the data stream. [9, p. 2–10] A routing unit can use the TDEST signal to deliver a packet to the the corresponding slave.

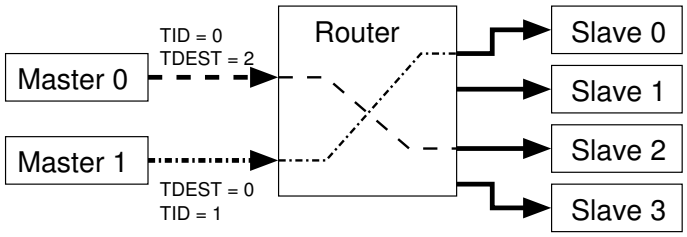


Figure 2.3: Example of routing packets among two masters and four slaves.

2.4 Reconfigurable System-on-Chip

As it was mentioned the System-on-Chip is a widely used type of device in the world of embedded systems. Such a device contains a processor that is running the main application and uses the application specific hardware to accelerate it or to interface with other external systems and peripherals. Because both the processor and the application specific hardware units are located on a single chip it saves the area, and possibly wires, on a PCB and enables to make the target system smaller. At the same time the integration reduces the power consumption and connections among processor and application specific hardware are shorter.

This work studies a family of SoC devices where the application specific hardware is reconfigurable. The term *Reconfigurable System-on-Chip* (RSoC) denotes this type of chips. Companies call their products of this kind as

- *System-on-Chip Field-Programmable Gate Array* (SoC FPGA)[1],
- *SoC FPGA with a Hard Processor System*[2], or
- *All Programmable SoC*[3].

An RSoC architecture consists of two parts. They are referred as *Processing System* (PS) and *Programmable Logic* (PL) in this text. This notation comes from the Xilinx's documentation of the Zynq platform (see [28, p. 23]). The PS contains a processor with basic peripherals and the PL represents the reconfigurable part of the chip.

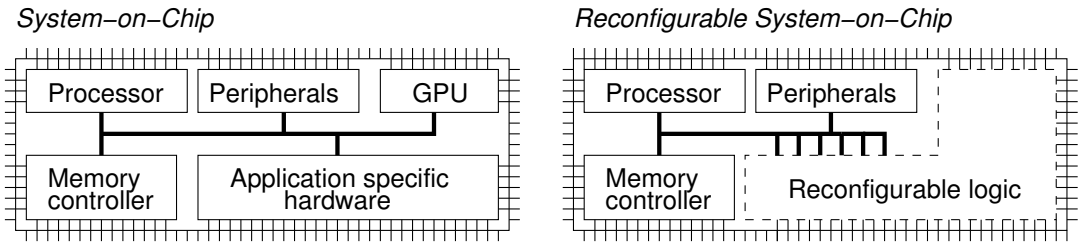


Figure 2.4: Comparison of a SoC and a Reconfigurable SoC.

The figure 2.4 summarizes the differences between both the classical SoC and the RSoC. The SoC provides high computation performance for application specific tasks (there are accelerators available), moderate performance for common and less common operations (computed by the processor). The SoCs for smartphones and other multimedia devices usually have a GPU to enable full HD 1080p video[4].

[1] `www.actel.com/fpga/SmartFusion2/`
[2] `www.altera.com/devices/fpga/cyclone-v-fpgas`
[3] `www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html`
[4] `www.nvidia.com/object/tegra-4-processor.html`, `www.ti.com/product/omap4430`, `www.qualcomm.co.in/products/snapdragon`, `www.broadcom.com/products/Cellular/Mobile-Multimedia-Processors/BCM2763`

The SoC devices are fabricated as ASIC chips. Once the chip is fabricated there is no way to modify its internals. For a different application that has other requirements, the used SoC must be exchanged for another. That leads to redesign of all the PCB because the chips are usually not pin compatible.

The RSoC offers great flexibility. A single chip can be used for different types of applications and if the PCB is general enough, redesign is not required. An already existing and working design can be reused by different configuration of the reconfigurable logic. Such a device provides high computation performance for application specific tasks and moderate performance for other common and less common operations. But if needed, the less common operations can be accelerated as well to provide the same level of performance as the critical parts of the application. The advantage of RSoC is the time to market. The time to design and test the application is lower and the developer can immediately see the hardware working. Because the system can be reconfigured after deployment a later bugfixes are also possible.

2.4.1 Potential of Reconfigurable System-on-Chip

The dynamically reconfigurable systems, such as FPGA or RSoC, offer many approaches to design embedded systems and computer systems in general. A great number of possible architectures have been developed and explored already by the industrial and scientific community. The leading reconfigurable platform are FPGAs. We can find many different FPGA devices on the market. They integrate the basic hardware logic such as Look-up Tables (LUTs), multiplexors and registers, but the modern FPGAs contain also fast memories, DSP units, and some of them also contain a processor. The RSoC is just a different point of view on the FPGA platform. The main difference is in the role of the processor part (the PS) in the system. For RSoC the processor is the most essential unit that runs the application and the FPGA is an additional component. [4, p. 3]

Reusing the area. The FPGA platforms provide a possibility of *partial dynamic reconfiguration* that enables use of one area of the chip for different tasks in time. This makes it possible to create smaller devices and to accomodate the software to choose the task to be computed in the hardware during runtime. The RSoC seems to be a perspective platform for such a type of computing. One of interesting scientific projects that implements this idea is BORPH, the Berkeley Operating system for ReProgrammable Hardware.

BORPH provides kernel support for FPGA applications by extending a standard Linux operating system. It establishes the notion of hardware process for executing user FPGA applications. Users therefore compile and execute hardware designs on FPGA resources the same way they run software programs on conventional processor-based systems. BORPH offers run-time general file system support to hardware processes as if they were software. The unified file interface allows hardware and software processes to communicate via standard UNIX file pipes. Furthermore, a virtual file system is built to allow access to memories and registers defined in the FPGA, providing communication links between hardware and software. [20, p. 2]

3 Xilinx Zynq

The Xilinx Zynq-7000 family is a System-on-Chip architecture that integrates a dual-core ARM Cortex-A9 MPCore based *Processing System* (PS) and Xilinx *Programmable Logic* (i. e. FPGA) in single device, built on 28 nm process technology. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes On-Chip Memory (OCM), external memory interfaces and a set of I/O peripherals. The Zynq offers the flexibility and scalability of an FPGA, while providing performance, power, and ease of use typically associated with ASIC and ASSPs. [28, p. 23]

The idea behind the Zynq is not new. There are FPGAs (e. g. the Xilinx Virtex-5 series [26]) that contain a PS part (usually a PowerPC processor) as an optional IP hardwired on the chip together with local memories (BlockRAMs), DSP blocks, AD converters, etc. The processor could be used for several complex tasks that would be difficult or costly to implement in the FPGA logic, consider for example DMA or TCP/IP communication. Such tasks require traversing through various data structures located in different locations of memory (linked lists, binary trees) or managing a complex state machines. But for some time critical applications these processors could be too slow. [5, p. 15]

However, the Zynq platform is different from the older approaches. The PS is considered to be the essential part of the chip and so it is possible to see Zynq as just a kind of an ARM SoC with an optional FPGA fabric. This approach is noticeable especially in the boot procedure (described in *3.1 Boot procedure*).

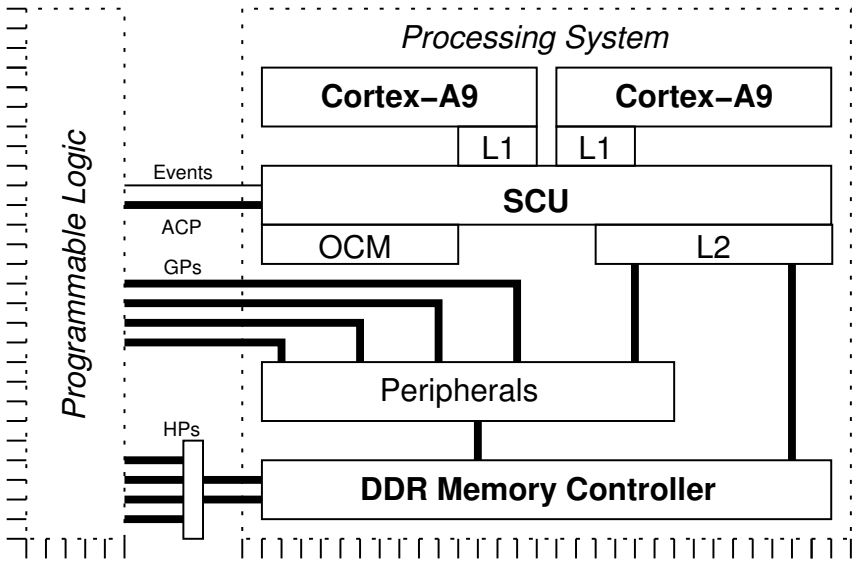


Figure 3.1: The Zynq architecture composed of the PS and the PL.

In the figure 3.1 there is a system view of the Zynq platform. You can see the processor cores with peripherals integrated inside the PS. There are controller peripherals of different types:

- connection of external memories: NOR, SRAM, NAND, SD, SDIO, MMC;
- general communication interfaces: USB, UART, SPI, Dual Quad SPI, I^2C, GPIO;
- the network interfaces: Gigabit Ethernet, CAN;
- and some other miscellaneous ones intended especially for debugging.

There are several external pins available exclusively to the PS, others that are available exclusively to the PL and several that can be shared between both of them.

3.1 Boot procedure

The booting of Zynq is done at least in two stages. When booting a Linux operating system a third stage is used. The Processing System boots first and then it may choose to setup the Programmable Logic. The system can work without completely without that. This was not possible with the previous FPGA centric devices.

3.1.1 Stage 0

The initial device startup is controlled by the zero booting stage (stage 0). The ARM CPU 0 starts executing non-modifiable code located in the BootROM memory after power-on reset. For the security reasons, the CPU is always the first device out of reset among all master modules within the PS. The purpose of the stage 0 is to load and execute the stage 1. There are three different PS configurations for this stage:

1. secure master mode,
2. non-secure master mode, and
3. non-secure slave mode via JTAG. [28, p. 140]

In either master mode the boot is done from one of the four supported master sources: NAND, NOR, Quad-SPI, or SD. [28, p. 143]

The stage 0 supports multi-boot, the capability to fall-back to a golden stage 1 image. The fall-back and golden images must be either both encrypted by the same key or both non-encrypted.

The secure master mode. The stage 0 loads a stage 1 executable image from the selected external source into the OCM. The image is decrypted by AES-256 algorithm and verified by HMAC SHA-256 during this process. Because the AES and HMAC engines reside in the PL part of the chip, the PL is required to be powered for the initial boot sequence (the stage 0 verifies that the PL has power before the decryption starts). After that, the stage 1 is executed. [28, p. 144–146, 171–172, 639–640]

If the device has been booted in the secure mode, the security policy block in the PS monitors the system status. When a conflicting status is detected a security lockdown

is triggered. In a security lockdown the OCM and all system caches are cleared, the PL is reset and the PS enters a lockdown mode. The only way to continue is a power-on reset. [28, p. 643]

The non-secure master mode. The stage 0 loads a stage 1 executable image from the selected external source into the OCM. The PL is not required to be powered on at this stage. Then the stage 1 is executed. [28, p. 170]

The non-secure slave mode. In this mode, the PS is a slave to the JTAG port. The PL's external JTAG pins are used, the PL is required to be powered up. The secure images are not allowed in this mode. [28, p. 172]

3.1.2 PL configuration

Unless the JTAG is used for booting, the PL must always be configured using the PCAP interface. Users are free to configure the PL at any time. It can be done either at PS boot or later. The PL must be powered on before it can be configured. The PL executes its power-on reset sequenece to clear all the PL's configuration SRAM cells. This procedure can be monitored by the PS. To configure the PL by a bitstream, the clear must be finished. After the PL has been configured, it can be reconfigured using either the PCAP or ICAP interfaces. To perform a secure configuration of the PL, the PS must have booted securely. The AES and HMAC engines can only be enabled by the stage 0. [28, p. 174–176]

3.1.3 Stage 1

The first stage bootloader (FSBL) is loaded from an external source by the stage 0 and executed. It is a user defined stage. It can load a bitstream into the PL, load another software, or start other processing. [28, p. 140]

3.2 Processing System

The Processing System is based on the dual-core ARM Cortex-A9 MPCore capable of asymmetrical or symmetrical multiprocessing configurations. Beside the CPUs, the MP-Core contains a Snoop Control Unit (SCU) that assures coherency within the cluster, a set of private peripherals (timers and watchdogs), and an interrupt controller. [28, p. 28], [13]

3.2.1 ARM Cortex-A9

ARM Cortex-A9 is a 32-bit RISC processor with Hardward Load/Store architecture. It provides sixteen 32-bit visible registers with mode-based register banking. As an extension a NEON (implementation of the SIMD instruction set) and VFP (Vector-Floating-Point Architecture supporting single and double-precision arithmetic) technologies are included in the Zynq platform. The processor supports both the big-endian and little-endian data access. [12, p. 2-8], [28, p. 28]

The ARM Cortex-A9 cores conform to the ARM ARMv7-A architecture where the v7 refers to version 7 of the architecture, while A indicates the architecture profile that describes Application processors. [12, p. viii]

There are three profiles defined by ARM:

1. **A**—the *Application* profile defines an architecture aimed at high-performance processors, supporting virtual memory system using a *Memory Management Unit* (MMU) and therefore capable of running complex operating systems. Support of the ARM and Thumb instruction sets is provided.
2. **R**—the *Real-time* profile defines an architecture aimed at systems that need deterministic timing and low interrupt latency and which do not need support for a virtual memory system and MMU, but instead use a simpler memory protection unit (MPU).
3. **M**—the *Microcontroller* profile defines an architecture aimed at low cost and low performance systems, where low-latency interrupt processing is vital. It uses a different exception handling model to the other profiles and supports only a variant of the Thumb instruction set. [12, p. 2-3]

The Cortex-A9 supports the following instruction sets:

- **ARM**. The standard 32-bit instruction set. [11, p. A1-3]
- **Thumb**. 16 or 32-bit instructions with a subset functionality of the ARM instruction set. It provides significantly improved code density, at a cost of some reduction in performance. [11, p. A1-3]
- **Thumb2**. An extension of the 16-bit Thumb instruction set to include 32-bit instructions. [11, p. A1-3]
- **Jazelle**. It is the Java bytecode execution extension. [11, p. A1-6]
- **ThumbEE**. A variant of the Thumb instruction set that is designed as a target for dynamically generated code. [11, p. A1-6]

3.2.2 Snoop Control Unit

The SCU is an interconnection among the Cortex-A9 cores, the Accelerator Coherency Port (ACP), and the memory system. It maintains the data cache coherency between the cores' L1 memories and serves the L2 memory access. There is no support for coherency of the instruction cache. The SCU allows to route memory-mapped access to one of its AXI master ports using the filtering capabilities. [13, p. 2-2, 2-13]

Note that the L1 caches for both Cortex-A9 cores are managed by the SCU.

The MPCore supports the MESI cache coherency protocol. In a correctly configured system, every cache line is dynamically marked with one of the following states:

- **Modified**—the data is present only in the current cache and it is dirty (not up to date with the next memory level).
- **Exclusive**—the data is present only in the current cache and it is clean (up to date with the next memory level).

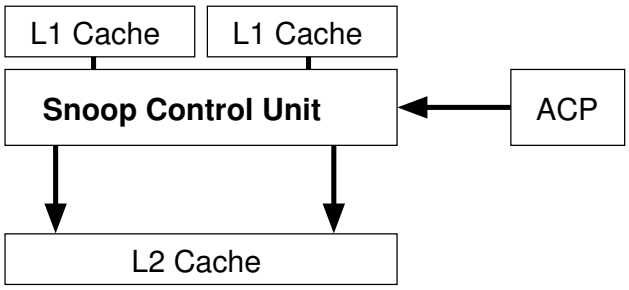


Figure 3.2: Snoop Control Unit.

- **Shared**—the data is present in one or more other core's caches and it is up to date.
- **Invalid**—the data is invalid. [10]

3.2.3 Event Interface

The cores provide an event interface that can be used for simple communication between a core and an external agent placed in the SoC. It is intended to be used in conjunction with the ACP. Any core can issue the `WFE` or `SEV` instruction. The agent can detect the execution of a `SEV` instruction by reading the `EVENTO` pin or wake up a core executing a `WFE` instruction by asserting the `EVENTI` pin. The interface allows to implement a semaphore. [13, p. 2-23]

3.2.4 On-Chip Memory and DDR Memory Controller

The main memory of the Zynq is divided between two parts: the On-Chip Memory and the DDR Memory. Both memories are accessible from either the PS and the PL. The On-Chip Memory is needed for the boot procedure (see *3.1 Boot procedure*).

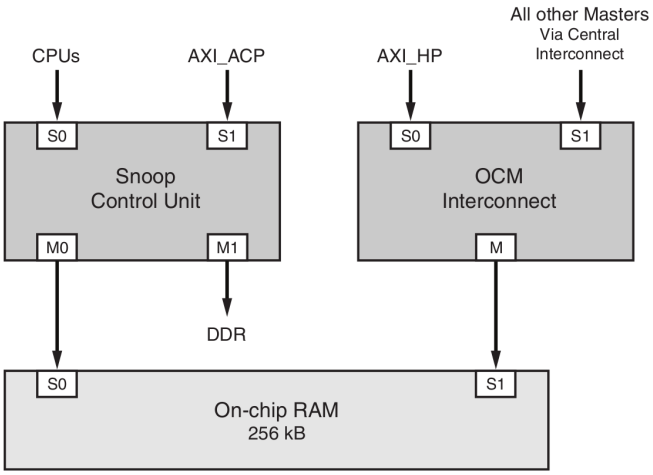


Figure 3.3: OCM interfaces schematic. [28, p. 619]

On-Chip Memory. The On-Chip Memory contains 256 KB of RAM and 128 KB of ROM (BootROM). It supports two 64-bit AXI slave interface ports, one is dedicated for CPU/ACP access and the other is shared by all other bus masters within the PS and the PL. The BootROM is used exclusively by the boot process and is not visible to the user. The entire memory can be divided into 64 4 KB blocks, and assigned security attributes independently (the TrustZone feature). The access into the OCM can be restricted by the Snoop Control Unit of the CPU. The access latency for CPU/ACP reads to the OCM is at least 23 cycles. [28, p. 617–618]

DDR Memory Controller. The DDR Controller available on the Zynq chip supports DDR2, DDR3 and LPDDR2 devices. There are four (duplex) 64-bit synchronous AXI interfaces available to serve multiple masters simultaneously. One port is dedicated to the L2 cache (for CPU cores and ACP port), two ports are dedicated to the AXI High-Performance ports (described in *3.3.2 High-Performance ports*) and the fourth port is shared by all other masters in the system. [28, p. 240]

The controller arbitrates requests from the eight simplex ports (four reads and four writes). The arbitration is based on a combination of

- how long the request has been waiting,
- the urgency of the request, and
- if the request is within the same page as the previous one. [28, p. 240]

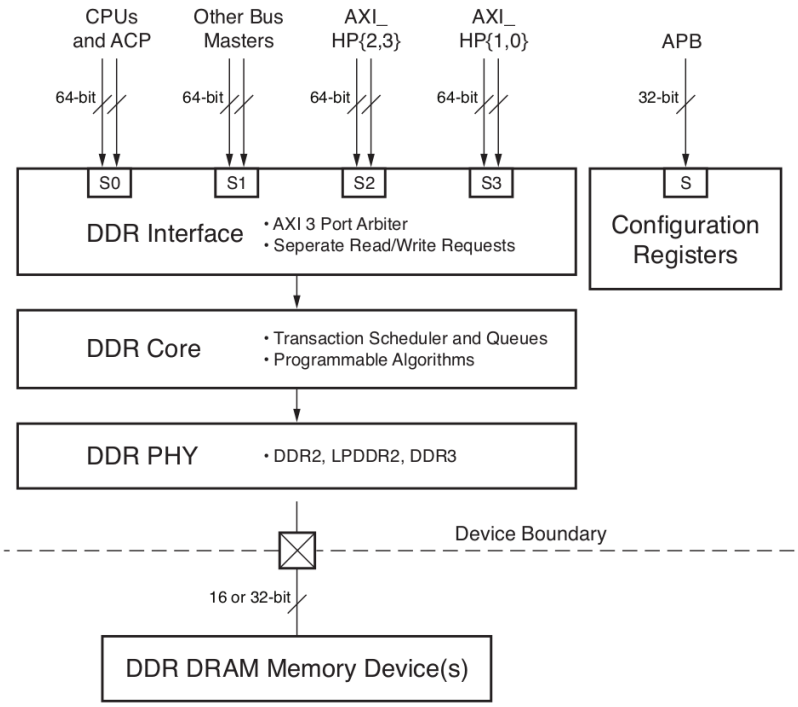


Figure 3.4: The three DDR Controller parts. [28, p. 242]

The controller is divided into three parts (as shown in the figure 3.4). The first part—DDR Interface—performs the arbitration. The DDR Core optimizes data bandwidth and latency by transaction scheduling and re-ordering. [28, p. 240–242]

Note that the maximum total memory density is 1 GB. [28, p. 244]

3.2.5 DMA Controller

The DMAC is integrated in the Processing System and uses a 64-bit AXI master interface. It is intended to perform DMA data transfers among system memories and peripherals without the processor intervention. The source and destination can be almost anywhere in the system. The memory map for the DMAC includes DDR, OCM, linear addressed Quad-SPI memory, SMC memory, and PL peripherals attached to the Master General Purpose AXI Interface. [28, p. 200]

The transfers are controlled by the DMA instruction execution engine. A program code for the DMAC is written by software into a region of system memory that is accessed by the controller. The controller can run up to eight channels—threads running on the DMAC's execution engine. [28, p. 200]

Each thread has its own program counter. A simple program for a thread may look this way:

```
1 DMAMOV CCR, SB2 SS32 SAI DB2 DS32 DAF
2 DMAMOV SAR, 0x1000
3 DMAMOV DAR, 0x4000
4 DMALP 16
5   DMALD
6   DMAST
7 DMALPEND
8 DMAEND
```

The line 1 configures:

- ARLEN = 2, ARSIZE = 32, ARBURST = INCR, and
- AWLEN = 2, AWSIZE = 32, AWBURST = FIXED.

Lines 2 and 3 set the source and destination address. The DMA transfer is performed by lines 4–7. The instruction `DMALP` defines a cycle that will loop 16 times. Each iteration performs a read burst (`DMALD`) from the source address and write burst (`DMAST`) to the destination address. In other words, the program would move $16 \cdot (2 \cdot 32)\ B = 1024\ B$ in 16 bursts from addresses 0x1000 up to 0x1400 to the fixed address 0x4000. There are instructions that can use the Event Interface described in *3.2.3 Event Interface*.

The DMAC exposes four request interfaces to the PL. This enables the PL logic to manage the flow of up to four DMA channels. [28, p. 201]

The estimated throughput of the DMAC is 600 MB/s. [28, p. 544]

Note that those peripherals are capable of *bus mastering* (i. e. perform DMA on their own): The Gigabit Ethernet Controller, SD/SDIO Peripheral Controller, USB Controller, and Device Configuration Interface.

3.3 Programmable Logic

The Programmable Logic is based on the 7 series Xilinx FPGAs. There are two families available:

1. Artix-7: a low cost and low power FPGA. It is shipped with the Zynq-7010 and Zynq-7020.
2. Kintex-7: a price-performance optimized FPGA. It is shipped with the Zynq-7030 and Zynq-7045.

Depending on the version, the PL offers 17,000–218,000 6-inputs look-up tables (LUT), 60–545 Block RAM memories with capacity of 36 Kb, 80–900 DSP slices, PCIe interface (just the Kintex-7), multi-gigabit transceivers and others. See the [21] and [27] for details.

PS-PL interfaces. There are five types of interfaces between the PS and the PL available in the Zynq:

1. Interrupts,
2. Event Interface,
3. General Purpose ports (GP),
4. High Performance ports (HP),
5. ACP. [28, p. 36–37, 53]

Each of them can be used for different types of communication.

3.3.1 General Purpose ports

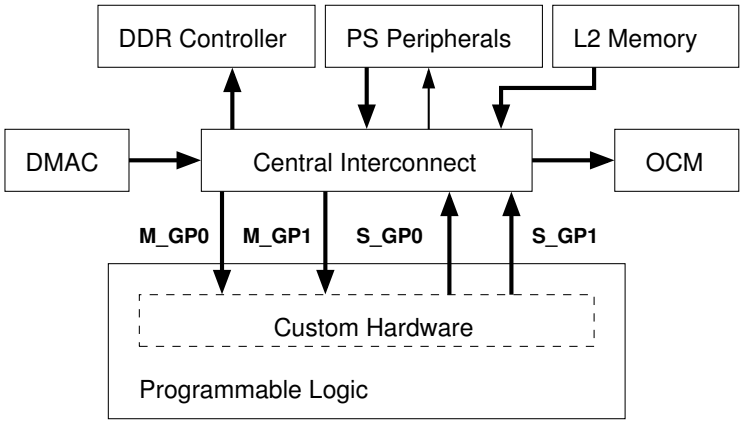


Figure 3.5: General-Purpose ports' connections (simplified). [28, p. 545]

The General Purpose (GP) ports consist of four 32-bit AMBA AXI3 ports—two slaves and two masters. The slaves allow devices to map into the PS address space. The masters can be used to access other peripherals, the memory, and OCM. These interfaces

are connected directly into the PS internal interconnect, without any additional buffering. The performance is constrained by the ports of the internal interconnect. These interfaces are not intended to achieve high performance. [28, p. 135]

The estimated throughput of each GP interface is 600 MB/s. [28, p. 544]

The master GP interfaces can be used by the DMA Controller that resides in the PS. It offers a moderate levels of throughput with little PL logic resource usage. [28, p. 545]

3.3.2 High-Performance ports

There are four 64-bit master High-Performance (HP) AMBA AXI3 ports available in the PL (also referenced as AFI—AXI FIFO Interface) that are connected into the main memory of the PS and into the OCM. The ports can be configured as 32 or 64-bit data wide master interfaces. Each port includes two FIFO buffers for read and write traffic. The PL could dynamically change the priority of the individual read and write requests. The FIFO levels can be used as a look-ahead to determine if data can be read or written without direct access to the AXI handshake signals. [28, p. 125–129]

When configured in the 32-bit wide mode, the user can control internal upsizing of the transactions. If a particular `AxCACHE`(1) = 1, the upsizing is done to make better use of the 64-bit bus available bandwidth. Only bursts multiplies of 2, incremental burst read commands, aligned to 64-bit boundaries are upsized. [28, p. 130–131]

The FIFOs are capable of multi-threaded out-of-order command processing and data beats interleaving. The DDR Controller guarantees that all read commands are completed continuously. However, it take advantage of re-ordering of read and write commands to perform internal optimizations. Therefore, sometimes the read and write commands can be completed in different order from which they were issued. [28, p. 132]

The estimated throughput of each HP interface is 1,2 GB/s. [28, p. 544]

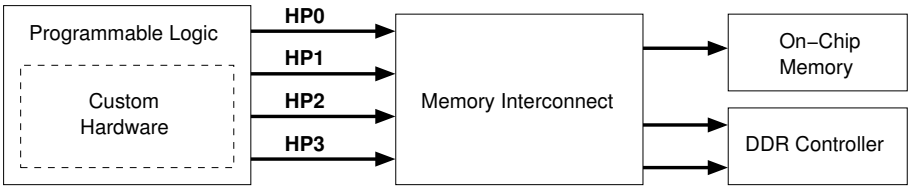


Figure 3.6: High-Performance ports' connections (simplified). [28, p. 127]

3.3.3 Accelerator Coherency Port

The ACP is a 64-bit AMBA AXI3 slave interface on the SCU (see figure 3.2). It provides cache-coherent access point directly from the PL to the processor subsystem. A range of PL masters can use the ACP to access the caches and the memory exactly the way the ARM cores do to simplify software, increase overall system performance, or improve power consumption. The ACP provides cache-coherent access while any memory local

to the PL are non-coherent with the ARM (because it would not be managed by the SCU). [28, p. 98–99], [13, p. 2-20]

The ACP interface can make following types of requests:

- ACP coherent read request—when $\texttt{ARUSER}(0) = 1 \wedge \texttt{ARCACHE}(1) = 1$.
- ACP non-coherent read request—when $\texttt{ARUSER}(0) = 0 \wedge \texttt{ARCACHE}(1) = 0$.
- ACP coherent write request—when $\texttt{AWUSER}(0) = 1 \wedge \texttt{AWCACHE}(1) = 1$.
- ACP non-coherent write request—when $\texttt{AWUSER}(0) = 0 \wedge \texttt{AWCACHE}(1) = 0$. [13, p. 2-20]

For maximum performance of the ACP transfers the following burst configurations are recommended.

- A wrapped burst of four doublewords ($\texttt{AxLEN} = 3 \wedge \texttt{AxSIZE} = 3$) with 64-bit aligned address, and all byte strobes set.
- An incremental burst of four doublewords with the first address corresponding to the start of a cache line, and all byte strobes set. [13, p. 2-21]

Accesses that do not match this format cannot benefit from the SCU optimizations, and have significantly lower performance.

ACP provides a low latency path between the PS and the accelerators implemented in the PL when compared with a legacy cache flushing and loading scheme. An acceleration can be done by the following steps:

1. The CPU prepares input data for accelerator.
2. The CPU sends a message to the accelerator through a General Purpose port or by an event.
3. The accelerator retrieves the data via the ACP, performs the appropriate computation, and returns the result back via ACP. The comunication is done by a DMA engine. When a cache hit occur, the access latency is small.
4. The accelerator notifies the CPU that the computation is complete by an interrupt or an event. [28, p. 99–100]

3.3.4 DMAC Interface

As it was mentioned earlier in section *3.2.5 DMA Controller*, the Programmable Logic is able to request DMA transactions. For that purpose there are four pairs of AXI-Streams, which can be associated with up to four channels of the DMAC. The DMAC must be programmed in the standard way. The instruction `DMAWFP` instructs a DMAC thread to wait until the specified peripheral requests for that DMA channel. The PL interface can request a DMA by using the `DMA_DR` interface and accept acknowledges via the `DMA_DA` interface.

3.4 Designing a PL accelerator

To implement an accelerator that offloads a computation into the Programmable Logic an application specific hardware unit is needed. To achieve high throughput a High-Performance Port or the Accelerator Coherency Port is utilized to connect the hardware unit to the system. To handle such a port the hardware unit needs to be a *bus master* because there is no central DMA unit available for those ports.

The software must provide a memory area to the accelerator to interchange the input and output data of the computation. The addresses and sizes are written into the accelerator to be able to perform the computation. Finally the software can see the result in memory. For this purpose, an operating system driver must be implemented.

3.4.1 Hardware accelerator design

The hardware accelerator is composed of two parts: *AXI4 Bus Master* and *Accelerator Engine*. Both parts are connected by AXI4-Streams that are not difficult to handle by the engine. The AXI4 Bus Master needs to be configurable by the driver in the operating system. For the Accelerator Engine the configuration is optional (depends on the application).

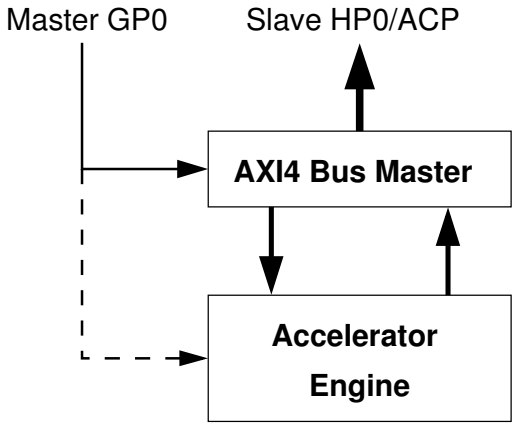


Figure 3.7: Hardware accelerator for the Zynq.

4 The Linux Kernel

The Linux Kernel is an open source operating system kernel. It supports a great number of computer architectures, including ARM. These two factors make it a good choice as a kernel for operating systems running on various embedded systems. When designing an application divided between HW and SW, developers may need to extend the used kernel to be able to communicate with the hardware part of the application. This chapter describes the most essential knowledge needed to extend the Linux Kernel.

4.1 Linux module

The Linux Kernel supports dynamic insertion and removal of code at runtime. Related subroutines, data, and entry and exit points are grouped together in a single binary image, a loadable kernel object, called a *module*. Support for modules allows systems to have only a minimal base kernel image, with optional features and drivers supplied via loadable, separate objects. Modules enable loading of new drivers on demand in response to the hot plugging of new devices. [2, p. 338]

When developing a device driver a module is always the most essential part of it. When the module's entry point is executed (after it is loaded into the running system) it registers a new device driver to the kernel and provides information about supported devices. The kernel is then free to use the driver if an appropriate device is connected.

4.2 Linux device drivers

The figure 4.1 shows the (simplified) model of structures used to represent a computer system in the Linux Kernel. Each physical device is represented by the corresponding `struct device` and has associated a number of other entities.

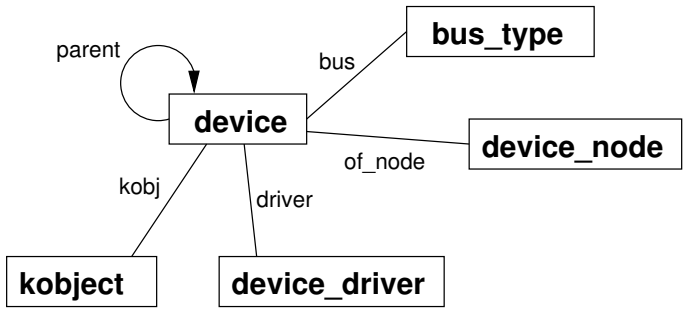


Figure 4.1: The model of device related structures inside the Linux Kernel.

The kernel needs to know how a device can be accessed. Each computer system is composed of various bus systems like PCI, USB, etc. to which the devices are connected.

Usually there is a bridge—either integrated inside the SoC or in the form of an external chip placed on the PCB—that serves as an intermediator to access such device. The way of accessing devices over a specific bus system is covered by an instance the `struct bus_type` that every device in the system must be associated with. The Linux Kernel provides predefined instances of the `struct bus_type` for each bus system it can use. For example: `usb_bus_type`, `pci_bus_type`, `spi_bus_type`, `i2c_bus_type`, and others. For devices that are connected directly to the CPU, and this is often the case of System-on-Chip architectures, a special bus system exists—`platform_bus_type`.

When a device is detected by the Linux Kernel core the information about the device is used to select a driver that will take care of it. This operation is called *probing*. Each device driver (represented by `struct device_driver`) defines a *probe function*. Its purpose is to assure that the kernel selected the correct driver and to initialize the driver instance's internals.

The way of acquiring information about devices varies among different computer architectures. The ARM architecture and few others utilize a device tree that describes the static structure of the computer system. This particularly incorporates bus controllers (PCI, USB, etc.), available CPUs, and memory. The Linux Kernel requires a set of minimal information about the system to be able to boot. The other kernel code (device drivers) are free to access the nodes of the device tree when probing for a device. Each device is associated to an appropriate node of the device tree (if it is specified there) by the `of_node` in the `struct device`.

At the heart of the device model there is the *kobject*, short for *kernel object*, which is represented by `struct kobject`. It is similar to the `Object` class in object-oriented languages such as C# or Java. It provides basic facilities, such as reference counting, a name, and a parent pointer, enabling the creation of a hierarchy of objects. The *sysfs*, a virtual user-space filesystem, is a representation of the kobject hierarchy inside the kernel. [2, p. 349–350]

4.3 Device tree

The *Open Firmware Device Tree*, or simply device tree, is a data structure and language for describing the topology of hardware at runtime. When using a device tree the OS does not need to hard code details of the underlying machine (this enables building of ARM multi-platform kernels). [16], [14]

The device tree originates in the Open Firmware [19] specification as a part of the data passing method to a client program (OS). Open Firmware is used on PowerPC and SPARC platforms. The infrastructure implemented for that purpose was generalized to be usable by all architectures. Currently, six mainlined architectures have some level of device tree support (x86, ARM, MicroBlaze, PowerPC, MIPS, and SPARC).

When booting a system, the bootloader passes a device tree binary to the kernel. The binary is a data structure suitable to be read by the OS. It can be created by a device tree compiler from a human-readable specification. Such a compiler is a standard part of the Linux Kernel build system. [16]

First, the kernel uses data in the device tree to identify the specific machine. The ARM architecture benefits from this approach because there are lots of different ARM-based System-on-Chip architectures. Each of them may have specifics the kernel needs to know. For that purpose, the *compatible* property in the root node of the device tree is used. The compatible property contains a sorted list of strings starting with the exact name of the machine and followed by an optional list of compatible boards (sorted form the most compatible to the least). [16]

In similar manner, the compatible property is used to identify hardware devices in the underlying computer system. The kernel matches the property with strings included in the device drivers. The device tree describes topology of the system, i. e. how are the devices connected to the processor. The nodes that are directly connected to the processor bus are identified as *platform devices*. For those devices, the kernel allocates and registers a `struct platform_device` instance (a specialization of `struct device`) that may get bound to a `struct platform_driver` instance (a specialization of `struct device_driver`). [16]

4.4 Writing character drivers

In Linux, as with all Unix systems, devices are classified into one of three types:

- block devices,
- character devices,
- network devices.

The *block devices* are addressable in device-specified chunks called *blocks* and generally support *seeking*, the random access of data. Examples of block devices include hard drives, Blue-ray discs, and memory devices such as flash. Block devices are accessed via a special file called a *block device node* and generally mounted as a filesystem (the special files are not accessed directly but through the virtual filesystem of the kernel). [2, p. 337]

The *character devices* are generally not addressable, providing access to data only as a stream, generally of characters (bytes). Examples of character devices include keyboards, mice, and printers. Character devices are accessed via a special file called a *character device node*. Unlike with block devices, applications interact with character devices directly through their device node. [2, p. 337]

The network devices are out of the scope of this work.

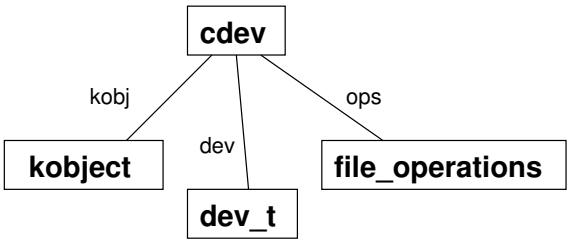


Figure 4.2: The representation of a character device inside the Linux Kernel.

A character device is represented by `struct cdev` in the kernel. As shown in the figure 4.2, an instance of `struct cdev` has an associated kobject to be manageable by the kernel, an identification represented by the `dev_t` data type and an *ops* member that is an instance of the `struct file_operations`.

Device node. Each character device is related to its device node, a special file in the filesystem, through a pair of numbers: *major* and *minor*. These numbers are stored in the *dev* member of every `struct cdev` instance. To access a character (or block) device from user space, the corresponding device node is a hint for kernel to select the right driver to handle the operations the user wants to perform. The *major* identifies the driver and the *minor* points to one specific instance of the driver. The available majors can be obtained by

```
$ cat /proc/devices
```

on any Linux station. A device node can be created using the *mknod* program

```
$ mknod /dev/mydevice c 254 0
```

where the letter 'c' denotes a character device, the number 254 is its *major* and the last value 0 is its *minor*. The current Linux systems provide tools to create the required device nodes automatically (using *devtmpfs* and daemons like *udev*) when a device is plugged into the system but only if the driver supports that. To check the major and minor of a special file, one may call e. g.

```
$ ls -l /dev/tty1
crw--w---- 1 root tty 4, 1 May 9 14:01 /dev/tty1.
```

Here the letter 'c' (at the beginning) denotes a character device and the two numbers 4, 1 are the major and minor.

In previous versions of the Linux Kernel (prior to the release 2.6.0) the majors were associated to the drivers statically. The file `Documentation/devices.txt` provides a list of assigned majors for common drivers. New major numbers are no longer being assigned. The Linux Kernel provides functions to allocate them dynamically at runtime. This effectively simplifies development of new device drivers for Linux. [15]

Operations. The `struct file_operations` defines a set of functions (operations) that can be performed on a character device. Implementation of a character device requires definition of at least a subset of those functions. The most notable are:

- `open`—requests to prepare the device driver for service,
- `release`—notifies that the user does not want to use the device anymore,
- `write`—writes a data piece into the device,
- `read`—reads a data piece from the device, and
- `mmap`—enables to map an address space of the driver (can be used to access the hardware device directly or for sharing a single buffer between kernel and userspace) into the address space of the user space program.

4.5 Memory allocations

The kernel uses various types of addresses a driver may need to handle. Each of them is related to the physical memory in some way:

- **Physical addresses** are the addresses the CPU use to access the system memory.
- **User virtual addresses** are regular addresses seen by the user-space programs. Each process has its own virtual address space. The kernel handles the translation into the physical addresses.
- **Bus addresses** are used between peripheral busses and memory. Often, they are the same as the physical addresses. Some architectures can provide an I/O memory management (IOMMU) that remaps addresses between a bus and main memory. In such a case, a configuration of an IOMMU must be done before setting up DMA operations.
- **Kernel logical addresses** represent the normal address space of the kernel. These addresses map some portion of main memory and are often treated as if they were physical addresses. On most architectures, logical addresses and their associated physical addresses differ only by a constant offset. These type of addresses are returned from `kmalloc()` allocation function.
- **Kernel virtual addersses** are similar to logical addresses, they represent a mapping from a kernel-space address to a physical address. All logical addresses *are* kernel virtual addresses, but many kernel virtual addresses are *not* logical addresses. The `vmalloc()` returns a virtual address. [1, p. 413–414]

Page allocation. The kernel has several ways for memory allocation. The lowest level mechanism has the memory page size granularity. It allows to allocate 2^{order} contiguous physical pages. The caller can be given either pointer to the list of allocated `struct page` (structure representing a physical page) instances or the kernel logical address. The core function of this mechanism is `alloc_pages(gfp_mask, order)`. The `gfp_mask` specifies how is the kernel supposed to allocate the requested memory. There are many different modifiers of this type, the most common are:

- `GFP_KERNEL`—represents a normal allocation and might block. Such an allocation can be performed only in the process context when it is safe to sleep (i. e. not when handling an interrupt).
- `GFP_ATOMIC`—such an allocation is of a high priority and must not sleep. This one is suitable to be used in interrupt context.
- `GFP_DMA`—requests to perform the allocation from `ZONE_DMA`. It is usually combined with other flags. [2, p. 235, 239, 241]

Because of hardware limitations, the kernel cannot treat all pages as identical. Some pages cannot be used for certain tasks (e. g. DMA by ISA peripherals). To overcome this limitation, the kernel divides pages into *zones*. The zones do *not* have any physical relevance but are simply logical groupings used by the kernel to keep track of pages. The kernel defines four primary memory zones:

- `ZONE_DMA`—This zone contains pages that can undergo DMA.
- `ZONE_DMA32`—Similar to `ZONE_DMA`, but those pages are accessible only by 32-bit devices.
- `ZONE_NORMAL`—The normal, regularly mapped, pages.
- `ZONE_HIGHMEM`—The so called "high memory" represents pages that are not permanently mapped into the kernel's address space. [2, p. 233–234]

Byte-sized physically contiguous allocation. A higher level approach for memory allocation is done by function `kmalloc(size, gfp_mask)`. This is the preferred interface in the kernel. The function returns a pointer to a memory region that is at least `size` bytes long. Note that the low-level allocations are page based. The call is similar to the `malloc()` function used in the user-space. [2, p. 238]

Virtually contiguous allocation. The physically contiguous memory area is needed especially when accessing a hardware device (e. g. DMA transfers). For other purposes the virtually contiguous memory can be sufficient. That means the memory seems to be contiguous but there is no guarantee that they are actually contiguous in physical RAM. Such allocations can be done by the function `vmalloc(size)`. The kernel code preferres `kmalloc()` over `vmalloc()` for performance reasons. [2, p. 244]

Slab layer. The slab allocator provides an efficient way of handling allocations for frequently allocated and deallocated data structures using a cache. The cache is divided into *slabs*, one or more physical contiguous pages. Each slab contains a number of *objects* which are data structures being cached. This strategy reduces fragmentation of memory and allows other optimizations. Interestingly, the `kmalloc()` is built on top of the slab layer. The interface of the slab allocator is out of the scope of this work. [2, p. 245–246]

Deallocating memory. Every allocation method has its counterpart to release the allocated memory. Unlike the user-space programs, whose memory space is managed by the kernel, a memory leak (a missing deallocation) or double deallocation of one memory chunk can result in serious problems of all the system. For the mentioned interfaces, the matching deallocation functions are: `__free_pages(page, order)`, `kfree(ptr)` and `vfree(addr)`. [2, p. 237, 238, 245]

4.6 Handling DMA capable devices

The Linux Kernel provides a common API for performing DMA operations. The API is accessible through the `linux/dma-mapping.h` header file. Any device (by means of the `struct device`) can be DMA capable if its hardware counterpart (a DMA controller) provides such functionality. In fact, no additional setup is necessary to make a device usable by the means of the API. To be more accurate, the kernel documentation states that for correct operation, the device driver must interrogate the kernel in the probe routine to see if the DMA controller on the machine can properly support the DMA addressing limitation the controller has. [17]

Query for support. To query the kernel whether the device is usable on the current machine, the driver calls `dma_set_mask(device, mask)`. The `device` represents the instance of `struct device` and the `mask` is a bit mask describing which bits of an address the controller supports. If the query returns zero, the device can perform DMA properly. Otherwise performing DMA will result in undefined behaviour. More masks can be queried this way, however, the most specific one must be the last. By default, the kernel assumes that the controller can address the full 32-bits address space. For *coherent mappings*, a similar call `dma_set_coherent_mask()` is provided. [17]

DMA'able memory. The memory that can be used for DMA can be obtained from various sources. Every physically contiguous cacheline-aligned memory can be used for DMA transfers. This means that memory obtained by the kernel allocators is DMA'able with the exception of `vmalloc()` call, where it is necessary to walk the corresponding page tables to obtain the physical addresses. [17]

DMA mappings. The Linux Kernel uses the dynamic DMA mapping. That is a combination of allocating a DMA buffer and generating address for that buffer that is accessible by the device. This includes configuration of IOMMU (if the platform contains one) or using of a *bounce buffer*. Bounce buffers are created when a driver attempts to perform DMA on an address that is not reachable by the peripheral device. This necessarily slows the process down because the data must be copied to and from the bounce buffer. DMA mappings must also address the issue of cache coherency. Any changes to memory in the CPU caches must be flushed out before a transaction occurs. [1, p. 445]

A DMA mapping can be of one of two types: *a coherent DMA mapping* or *a streaming DMA mapping*. The latter one provides a buffer that is simultaneously available to both the CPU and the target peripheral. As a result, coherent mappings must live in a cache-coherent memory. Such a mapping can be expensive to set up and use. The former one, the streaming DMA mapping, is more under the control of its user who must explicitly set up the mapping for a single operation. [1, p. 446], [17]

The kernel developers recommend use of the streaming mapping over coherent mappings whenever possible. On some platforms, this kind of mapping can be optimized in ways that are not available to coherent mappings. The coherent mappings, which have a long lifetime, can monopolize the mapping registers (when using scatter-gather lists) for a long time, although they are not being used all the time. [1, p. 446], [17]

Coherent DMA mapping. Such a mapping is set up by calling to the function `dma_alloc_coherent()`. This function handles both the allocation and the mapping of the buffer. The function returns two addresses, the virtual kernel address for the driver and the associated bus address suitable to be used by the DMA controller. When the buffer is no longer needed, it should be released by `dma_free_coherent()`. The driver can now access the DMA buffer as usual. Another approach is possible for small coherent DMA mappings. The kernel contains a concept of *DMA pools*, a mechanism to manage a pool of small buffers of predefined size per a pool. It is implemented on top of the slab layer. [1, p. 446–448], [17]

Streaming DMA mapping. This type of mapping can be used with any DMA'able memory that was already allocated. To setup a streaming mapping the kernel needs to know the direction of the following DMA transaction, i. e. memory-to-device or device-to-memory. For that purpose, the following constants are defined: `DMA_TO_DEVICE` and `DMA_FROM_DEVICE`.

A mapping is set up by calling to `dma_map_single()` which maps a single buffer for a DMA transfer. The function returns the corresponding bus address suitable to be used by the DMA controller. Here a bounce buffer can be created by the DMA subsystem if necessary. After the DMA transfer is complete, the mapping can be deleted by `dma_unmap_single()`. [1, p. 448–449], [17]

For the *Scatter-Gather DMA* transfers another approach can be used. This type of DMA transfers is not represented by a single address and size, but it uses a list of entries referred as a scatter-gather list or only a scatter list. It contains entries describing the individual buffers available for the DMA transfer. A scatter list is mapped by the function `dma_map_sg()`. The implementation can merge several entries into one which can help when a DMA controller has a limited number of scatter-gather entries or cannot do scatter-gather DMA. After the mapping is done, the caller is intended to walk over the (possibly modified) scatter list and setup the pairs (address, size) into the hardware (this is a device specific operation). After the DMA transfer is finished, the scatter list should be unmapped by `dma_unmap_sg()`. [1, p. 450–451], [17]

A call to unmap expects the same arguments that were passed to the corresponding map operation. On many platforms, the `dma_unmap_single()` does nothing. Therefore, keeping track of the arguments can waste space. For such a purpose, there are macros to provide a portable way for defining the storage of the needed arguments: `DEFINE_DMA_UNMAP_ADDR` and `DEFINE_DMA_UNMAP_LEN`; and the corresponding accessors: `dma_unmap_addr_set` and `dma_unmap_len_set`. On the platforms, where unmap call does nothing, the macros are defined to be empty. [17]

Background of the DMA mappings. Each platform should provide an implementation of function `get_dma_ops(device)`. This is an internal API of the kernel. The call returns a pointer to `struct dma_map_ops` that defines a set of DMA mapping operations. Most architectures provide a predefined instances of that structure and returns a general one as default. However, each platform can implement a logic to determine which instance of `struct dma_map_ops` to provide. On the ARM architecture, every instance of `struct device` can hold a pointer to an instance of `struct dma_map_ops` that overrides the default one. This approach can be used for different requirements or possibilities of various DMA controllers.

To take advantage of the Accelerator Coherency Port a non-default set of operations can be used. For that purpose there is an undocumented ARM-specific set of DMA operations `arm_coherent_dma_ops` inside the kernel. More information about this topic can found in the Git history of the Linux Kernel mainline[5].

[5] The related commits are: *v3.6-5693-gca41cc9*, *v3.6-1916-g7368a5d*, *v3.6-1915-g038ee8e*, *v3.6-1914-g4e770b2*.

4.7 Interrupt handling

The kernel can be interrupted at any time to process interrupts from peripherals. Different devices can be associated with different interrupts by means of a unique value associated with each interrupt. This enables the operating system to differentiate between interrupts and to know which hardware device caused which interrupt. In turn, the operating system can service each interrupt with its corresponding handler. The interrupt values are often called *interrupt request* (IRQ) lines. Each IRQ line is assigned a numeric value. Depending on the platform, some IRQs can be fixed and others assigned dynamically at runtime. [2, p. 113–116]

The response to a specific interrupt is called an *interrupt handler* or *interrupt service routine* (ISR). In Linux, an interrupt handler is defined as a C function with the following prototype:

`irqreturn_t irq_handler(int irq, void *data).`

It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. The primary goal of the handler is to acknowledge the interrupt's receipt to the hardware. The interrupt handler is executed in a special context called *interrupt context*. The code executing in this context is unable to block. As a result, calls to kernel functions that may sleep or block leads to undefined behaviour of the system. [2, p. 114–115]

Deferring work. Sometimes, an interrupt handler needs to perform a large amount of work. For such a purpose the interrupt handling is in general divided into two parts: *top halves* and *bottom halves*. A top half is the interrupt handler itself. It is expected to run immediately upon receipt of the interrupt and performs only the time-critical work. The rest of the execution is deferred until the bottom half. In the Linux Kernel, a bottom half can be implemented as a *softirq*, a *tasklet*, or a *work queue*. The first approach is reserved for the most timing-critical and important bottom-half processing. Currently only two subsystems of the kernel—networking and block devices—directly use softirqs. The preferred way is to use the tasklets. [2, p. 136,142]

Tasklets. Tasklets are built on top of softirqs. The kernel guarantees that only one tasklet of a given type runs at the same time. Note that tasklets cannot sleep. The interrupt handler schedules a tasklet by calling `tasklet_schedule(tasklet)`. A tasklet is represented by an instance of `struct tasklet_struct` and a C function with the following prototype:

`void tasklet_handler(data).` [2, p. 142–145]

Work queues. When a bottom half needs to sleep during its execution, the third approach must be used. A work queue executes in the context of a kernel thread. The kernel manages a default set of so called *worker threads* that handles the deferred work from multiple locations. A driver can use either a default thread or create its own thread to perform the work. A performance-critical work might benefit from the former approach. A work executed by a work queue is represented by an instance of `struct work_struct`

and a C function with the prototype:

```
void work_handler(void *data).
```

A work can be scheduled by calling `schedule_work(work)`. [2, p. 149–153]

The interrupt handling is a complex topic and cannot be covered in this text in detail. For more information, please, refer to the literature [2], [1].

4.8 Linux based operating system

To build an operating system based on the Linux Kernel for an embedded system, one needs to complete the following steps:

1. Obtain a toolchain containg the needed cross-compilers, linkers and basic libraries. This can be generally done in two ways: get a prebuilt binary toolchain or download the source code and build it. The GNU toolchain is mandatory to compile the Linux Kernel.
2. Compile the Linux Kernel with the obtained cross-compiler. This step needs to configure the kernel in a way that is specific for the target platform. For every supported architecture, a default configuration is provided by the kernel in the `arch/$ARCH/configs` directory (the `$ARCH` states for the target architecture). For the Zynq architecture, the setup can be

   ```
   $ make ARCH=arm xilinx_zynq_defconfig.
   ```

3. Prepare an image of the root filesystem suitable for the target platform. This step includes building of the application software and libraries by the cross-compiler and creation of the so called "init scripts" (when using the System V) and basic configuration files.
4. Compile a Linux-aware bootloader for the target platform. For this purpose the U-Boot bootloader is usually used. It provides support for various platforms and it can boot from many standard sources. The most important is its ability to boot the Linux Kernel.

All these steps can be very complex and can fail easily. It is also undesirable to do all the steps manually by hand. In the world of personal computers the Linux systems are prebuild by an organization to form a distribution consisting of the kernel, the bootloader, and the root filesystem (usually installed on a hard disk). The embedded systems are application specific and for them it is likely to be composed of as few applications and files as possible.

For that purpose a meta-distributions can be found on the Internet. A meta-distribution is a system that can build all the necessary parts of the operating system and lets its user to select what exactly to include inside. One of such meta-distributions is Buildroot[6]. It

[6] `http://buildroot.org/`

provides a configuration system *kconfig*[7] that enables to select various parameters of the target system. It can build or obtain a GNU toolchain for the target platform, build the Linux Kernel, and pack the selected user-space libraries and applications into a filesystem image. The usage of such a system simplifies the development significantly.

4.8.1 Das U-Boot

Das U-Boot[8] is the official name of the U-Boot bootloader intended for booting Linux Kernel in embedded systems. It provides a rich set of drivers for various peripherals, a runtime shell for service purposes and enables to boot from different sources like TFTP, SD memory card, and many others. This brings one disadvantage, the compiled U-Boot binary is quite big (few hundreds KB) for embedded systems. The size of the binary determines it to work as a second or third stage bootloader because it usually cannot fit into on-chip memories.

The bootloader provides an interactive shell that enables to customize the booting process without recompilation. Moreover, it is possible to create scripts to automate various booting scenarios while testing. In fact, the default booting sequence, the U-Boot executes automatically, is just a script hard coded into the binary. Booting from SD card with FAT filesystem is performed by the following steps:

```
1 > mmcinfo
2 > fatload mmc 0 ${kernel_addr} uImage
3 > fatload mmc 0 ${rootfs_addr} urootfs.cpio.gz
4 > fatload mmc 0 ${dtb_addr} zynq-zed.dtb
5 > bootm ${kernel_addr} ${rootfs_addr} ${dtb_addr}
```

On line 1, the SD card driver is initialized. Lines 2–4 loads three files into memory at addresses specified by environment variables of U-Boot. The variables can be set by `setenv` command, e. g.:

```
> setenv kernel_addr 0x1000000
```

The addresses must be chosen carefully to avoid overwriting of the loaded images by the `bootm` command. On line 5, the booting starts. The command `bootm` accepts address of the kernel in memory, then the address of the initial filesystem (can be omitted), and the last entry is address of the device tree binary in memory. U-Boot then reads the headers at the beginning of the loaded files `uImage` and `urootfs.cpio.gz` files to gather metadata useful for further processing. All three structures are moved to different locations and then the kernel is executed. The target location of the kernel is read from the U-Boot header, the location of the ramdisk is determined automatically while considering the environment variable `initrd_high`.

The U-Boot header for `urootfs.cpio.gz` (and also for custom scripts) can be created using `mkimage` command distributed with U-Boot. To create the `uroot.cpio.gz` file from `root.cpio.gz` for an ARM CPU, one may call

```
$ mkimage -A arm -T ramdisk -C gzip -d root.cpio.gz uroot.cpio.gz
```

[7] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
[8] http://www.denx.de/wiki/U-Boot

A script is packed in similar way:

```
$ mkimage -A arm -T script -C none -d script.txt uscript.bin
```

It can be executed by U-Boot on target platfrom by the `source` command after it is loaded into memory.

```
1 > fatload mmc 0 0x4000 uscript.bin
2 > source 0x4000
```

4.8.2 Buildroot on Zynq

For the purpose of this work a Linux operating system has been created using the Buildroot meta-distribution. The used Linux Kernel *xilinx-v14.5* comes from Xilinx's Linux repository[9]. This version contains code that is not in the mainline but provides support for Xilinx peripherals' drivers and other support for the Zynq platform. The used board was Zedboard[10] which contains the Zynq-7020. The kernel *xilinx-v14.5* contains device-tree support for this board.

The important configuration entries for Buildroot are:

```
01 BR2_arm=y
02 BR2_cortex_a9=y
03
04 BR2_TOOLCHAIN_EXTERNAL=y
05 BR2_TOOLCHAIN_EXTERNAL_CODESOURCERY_ARM201203=y
06
07 BR2_LINUX_KERNEL_CUSTOM_GIT=y
08 BR2_LINUX_KERNEL_CUSTOM_GIT_REPO_URL="git://github.com/Xilinx/linux-xlnx.git"
09 BR2_LINUX_KERNEL_CUSTOM_GIT_VERSION="xilinx-v14.5"
10 BR2_LINUX_KERNEL_DEFCONFIG="xilinx_zynq"
11 BR2_LINUX_KERNEL_DTS_SUPPORT=y
12 BR2_LINUX_KERNEL_INTREE_DTS_NAME="zynq-zed"
13 BR2_LINUX_KERNEL_UIMAGE_LOADADDR="0x8000"
14
15 BR2_TARGET_ROOTFS_CPIO=y
16 BR2_TARGET_ROOTFS_CPIO_GZIP=y
17 BR2_TARGET_GENERIC_GETTY_PORT="ttyPS0"
18 BR2_ROOTFS_DEVICE_CREATION_DYNAMIC_MDEV=y
```

Lines 1–5 select to compile for the ARM Cortex-A9 and define the toolchain. A prebuild binary toolchain CodeSourcery Lite would be used.

Lines 7–9 instructs Buildroot to download the Xilinx kernel from the specified Git repository On line 10, the kernel configuration is selected to be

```
arch/arm/configs/xilinx_zynq_defconfig
```

[9] `https://github.org/Xilinx/linux-xlnx.git`
[10] `http://zedboard.org`

and uses the version *xilinx-v14.5*. Lines 11–12 select the source for device tree blob generation

`arch/arm/boot/dts/zynq-zed.dts`.

The Linux Kernel build system creates automatically a kernel image with prepended U-Boot header. The used kernel version (since 3.7) builds multi-platform relocatable images for ARM (disassembling of the image shows that it is compiled with base 0xc0000000, however, the addresses are updated on startup by the kernel itself). The kernel should be able to boot on various ARM processors. Line 13 specifies where should U-Boot place the kernel in memory. The kernel relocates itself when booting to match this base address. This information is placed into the U-Boot header.

Configuration of the filesystem follows on lines 15–18. The filesystem generated by Buildroot would be in the CPIO format, i. e. a ramdisk, compressed by the GZIP algorithm. The default service console uses USB UART. It is represented by the `/dev/ttyPS0` character device in the kernel. The `/dev` directory would be managed by the system automatically.

After the configuration step is done (using the listed entries and `$ make menuconfig`) the operating system can be build by calling `$ make`. The results are located in the directory `output/images` (relative to the root of Buildroot source code). For Zynq, the following files are to be copied to an SD card:

- `uImage`—the Linux Kernel prepared for booting by U-Boot,
- `urootfs.cpio.gz`—the root file system (note that Buildroot cannot prepend the U-Boot headers in current version, so it must be done manually by calling to `mkimage`),
- `zynq-zed.dtb`—the device tree binary.

The created operating system can be booted on the board now.

It is possible to compile U-Boot for the Zedboard using the following Buildroot configuration entries:

```
1 BR2_TARGET_UBOOT=y
2 BR2_TARGET_UBOOT_BOARDNAME="zynq_zed"
3 BR2_TARGET_UBOOT_CUSTOM_GIT=y
4 BR2_TARGET_UBOOT_CUSTOM_GIT_REPO_URL="git://github.com/Xilinx/u-boot-xlnx.git"
5 BR2_TARGET_UBOOT_CUSTOM_GIT_VERSION="xilinx-v14.5"
```

However, the precompiled U-Boot distributed with Zedboard is suitable as well.

5 RSoC Framework

This work introduces a framework that consists of a set of hardware components and software drivers and provides a consistent extendable system for designing applications.

To design a system on a Reconfigurable System-on-Chip (RSoC) architecture the designer needs to implement the hardware adapter, connect it to the processor and write an appropriate software driver. However, most custom hardware adapters can be accessed in a generic way by means of data streams into the adapter and back into the software. For that purpose, a software driver and a hardware controller can be written just once to simplify the development of the application specific parts.

5.1 Concepts and goals

There are three general areas that the framework should cover. These are based on what a developer needs during the development of an application divided between hardware and software:

- Providing **reusable components** is a common and good practice used for development to simplify very common tasks like interfacing with complex bus systems, and solving simple but error prone problems.
- **Infrastructure** generation simplifies interconnection of adapters and accelerators with the Processing System. The bus systems consist of hundreds of wires that must be connected correctly. Automation is a desirable solution in this area.
- **Integration** of the hardware and software together is the essential goal of this work. It is possible to build software drivers and hardware controllers that can talk to each other and provide well defined interfaces to user specific hardware and software. Developers can take advantage of possibility to easily select and use a specific (DMA) controller for a particular hardware accelerator to trade-off between resources consumption and throughput.

5.2 RSoC Framework Overview

The main idea of the provided framework is to hide specifics of the target platform and enable the application code to use simple data streams between software and hardware parts. Applications use the standard system calls of the operating system—especially `write`, `read`—to send/receive data to/from the accelerator. The accelerators are connected to the software through a couple of simplex streams (one for each direction). The communication protocol defines a PDU called a *frame*. Both the software and hardware parts exchange frames among each other. The accelerator does not have to be aware of the particular implementation that ensures data moving between the Processing System and the Programmable Logic. The same applies to software. The framework inserts the

RSoC Bridge layer in between. The layer consists of hardware controllers and software drivers.

Frame. Each frame contains a very short *header* that is four bytes long. The header contains the size of the frame's *payload*. This rule limits the frame size to 4 GB. If longer frames are required, the future versions of the framework would extend the header.

RSoC Accelerator. The RSoC Framework defines an accelerator as an interface. In fact, one accelerator can use more RSoC Accelerator interface instances when needed. Each accelerator can have as many other interfaces that are not related to the framework as necessary. And there is a last point to mention, the term *accelerator* in the context of the framework does not necessarily mean a hardware unit that accelerates a function but it can be any kind of hardware unit. The most significant not-accelerating hardware units are adapters that helps to interface with another (external) system.

An RSoC Accelerator consists of four interfaces:

1. configuration bus (optional),
2. input data stream (for modes *Read-Only* and *Read-Write*),
3. output data stream (for modes *Write-Only* and *Read-Write*),
4. information vector.

The information vector (32 B long) helps to identify what accelerator is connected at the position. Currently, its structure is application dependent, but the last byte must be zero. In the future versions of the framework it will be standardized.

If the configuration bus is enabled, an address space range can be provided to the accelerator. Its base address is computed by the framework. The accelerator sees all bus transactions starting at address zero.

RSoC Accelerator to accelerate things. The purpose of the RSoC Accelerator interface is to increase throughput of a part of a software application. Such a unit performs a computation on an input data stream while generating an output data stream. The unit can require a configuration before the processing (load a cryptographic key, setup decision tables, etc.).

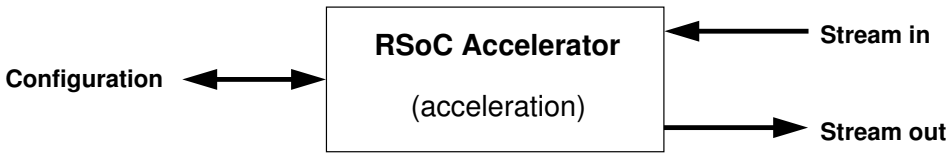


Figure 5.1: RSoC Accelerator (accelerating) interface.

The *Configuration* interface is the AXI4 bus (either AXI4-Full or AXI4-Lite). It enables access to the accelerator's custom address space. The AXI4-Lite variant can save resources on the chip. The *Stream in* and *Stream out* interfaces are the AXI4-Stream busses with

extended semantics. The streams consist of signals `TDATA`, `TKEEP`, `TVALID`, `TREADY`, `TLAST`, and `TUSER`.

The first data transfer (the first data beat) carries the frame size in the `TUSER` signal. Semantics of the `TUSER` in all other beats is not defined and can be used for any other purposes. Care must be taken when an AXI-Stream upsizer or downsizer is inside the data path because it may discard some `TUSER` values. However, the first `TUSER` value is always delivered. The usual width of the `TUSER` signal is 16 to 32 bits. There is one exception when the `TUSER` signal can be omitted. That is when the receiving accelerator does not need to know the frame size in advance.

RSoC Accelerator as an interface adapter. As it was mentioned the RSoC Framework recognizes the *interface* of an accelerator. The real purpose of the hardware unit hidden behind that interface can be an adaptation of the application to a particular physical interface that is not provided by the hardwired Processing System.

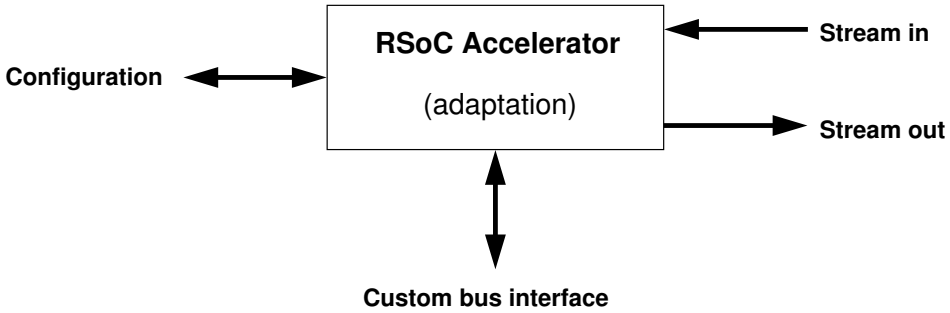


Figure 5.2: RSoC Accelerator (adapting) interface.

The interface of an adapter required by the RSoC Framework is exactly the same as for an accelerator. The adapter is expected to have another custom bus interface that is not under control of the framework.

RSoC Bridge. The infrastructure part of the system is implemented by the RSoC Bridge component. The component connects the RSoC Accelerator compatible units to the Processing System of the platform. It ensures that data coming from the software will be delivered to the accelerator and the data going out of the accelerator will be delivered to the software. The counterpart of the RSoC Bridge is a driver inside the operating system that provides similar services to the software part.

The number of slots for accelerators and Processing System interfaces is configurable and limited by the available resources on the chip.

RSoC Drivers. The framework needs to provide an interface for the userspace applications. For such purpose a communication protocol must be defined. The system is based on the Linux Kernel and that brings the first part of the software interface: character devices. Each accelerator is represented by one character device. To match the RSoC

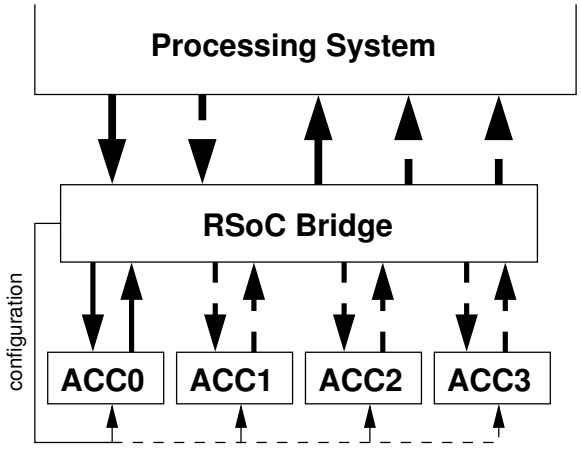


Figure 5.3: RSoC Bridge interface (dashed lines mark generic parts).

Accelerator interface the communication is based on *frames*. The frame header is part of the data stream. The driver expects the first four bytes to contain the frame size.

5.3 Reusable components

The first part of the RSoC Framework are components that are used by the framework itself, however, a designer can benefit of them when implementing hardware accelerators. Each component solves one particular issue related to the hardware designing but also simplifies design in the VHDL language. Some of the units are as simple as few interconnected wires while others contain automatons and other sequential and combinatorial logic. The components are grouped into few categories: AXI components, AXI-Stream components, general purpose components, and function packages.

Because the components can be optimized for different target platforms or another more efficient implementation can be provided by a third party, it is convenient to describe the components from the interface point of view. The RSoC Framework provides basic implementations (usually called *custom*) of all the components.

5.3.1 AXI components

The main infrastructure provided by the RSoC Framework uses three types of components: AXI 1-to-N, AXI N-to-1, and AXI Remap. The AXI 1-to-N provides interconnections between a one master device and N slave devices and the AXI N-to-1 interconnects N master devices and one slave device. The AXI Remap unit creates fixed interconnects among channels. Another unit required by the infrastructure is an endpoint of the AXI4-Lite bus. The communication is transformed there from the AXI4 channels into a register access logic.

AXI 1-to-N. The main purpose of the unit is to divide address space occupied by the slave devices. The component routes requests coming from the one master device.

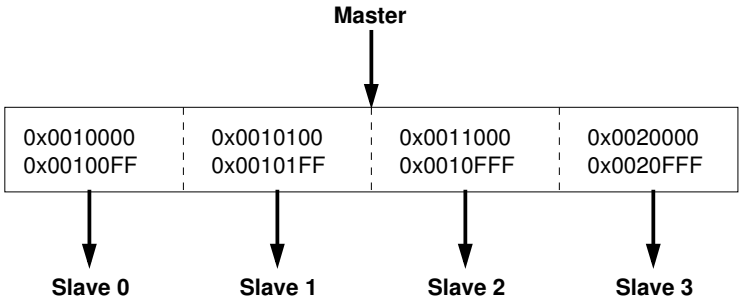


Figure 5.4: Component AXI 1-to-N schematic.

The routing is done using an address (coming from the master when requesting a new transaction) and address space description given during the hardware synthesis phase (so it is hardwired inside the component). Different implementations of this component can have various requirements and limitations and provide various features. There at least two basic parameters to consider:

1. what AXI protocol to route, and
2. what is the address space layout.

The implementation can support any of the three AXI protocols: AXI4, AXI4-Lite and AXI3. The address space layout is generated out of the component (it can be either hardwired or generated from another specification) and it must meet requirements of the implementation. The address space layout generation is defined in the Platform Package (see *5.3.4 Function packages*) so it is possible to change the generating algorithm based on the selected implementation of the component.

AXI N-to-1. The component enables the sharing of one AXI bus by many master devices. Its purpose is to arbitrate which master can communicate. The provided *custom* implementation is based on the arbiter component (see figure 5.9) that provides a simple round-robin decision logic. Another arbiter can be provided. The only important configuration is the AXI protocol as described for AXI 1-to-N.

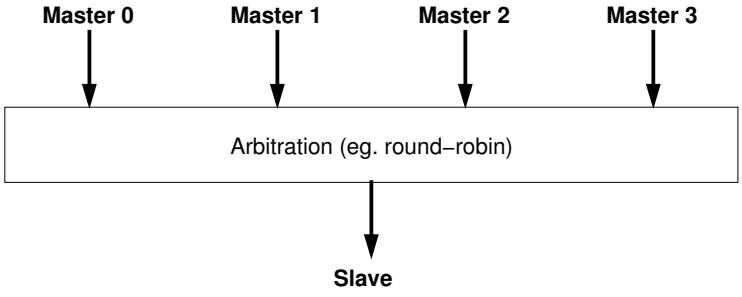


Figure 5.5: Component AXI N-to-1 schematic.

AXI Remap. The component is a set of wires (without a real logic) that interconnects a list of incoming AXI4 channels with the list of outgoing AXI4 channels. The main purpose

is to simplify VHDL coding (to overcome limitations of the language) and to increase readability. The component is given an array of numbers representing a mapping from the incoming busses to the outgoing busses and it connects the specfied pairs together.

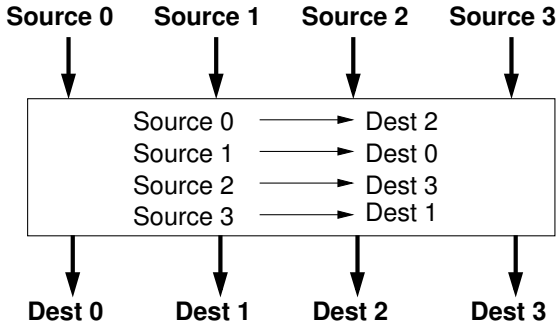


Figure 5.6: Component AXI Remap with a mapping specification as an example.

AXI Lite Endpoint. The AXI4-Lite interface implementation is time consuming and error prone because it needs to process five almost independent channels to access few registers of a component. The component converts the AXI4-Lite bus into two channels, the first one for reading registers and the second one for writing registers.

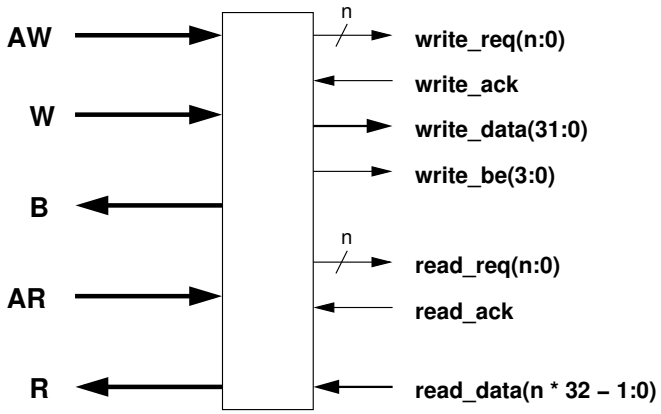


Figure 5.7: Component AXI Lite Endpoint serving n registers (each 32 bits wide).

5.3.2 AXI-Stream components

AXI-Stream busses are the basic building blocks of the processing using the RSoC Framework. It is desirable to have components that implements some general operations on the streams.

AXI-Stream FIFO. A queue (FIFO) is one of the most fundamental components for the hardware development. Its purpose is to buffer a piece of data or to divide

asynchronous hardware blocks. The RSoC Framework delivers an implementation based on the OpenCores GH[11] VHDL library.

AXI-Stream Start of Frame. The AXI-Stream data is usually transferred on the frame basis as described in *2.3.4 AXI4-Stream protocol*. The protocol does not provide any signalization to detect the beginning of a frame. For that purpose a simple automaton recognizing the first data transfer after the TLAST high signal can be constructed. Such a data transfer is marked as Start of Frame and it is the output of the component.

AXI-Stream Capture. The component captures data at a given fixed offset from the begining of each frame. The captured data is delivered through a stream to be processed by an external logic.

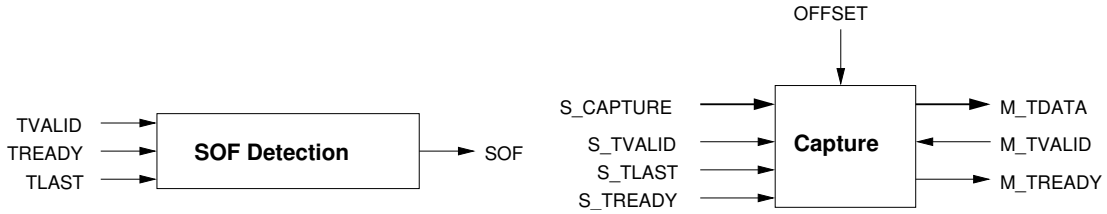


Figure 5.8: Start of Frame (left) and Capture (right) components' schematics.

AXI-Stream Discard. The component removes a frame out of the input stream when requested. Every frame that passes through the AXI-Stream Discard unit is first blocked until a command to *discard* or *transmit* the frame is received from an external logic.

5.3.3 General purpose components

General issues are solved by this group of components. It contains the essential units like *Generic Multiplexor* and *Onehot Decoder*, however, there are other general tasks the framework needs to solve.

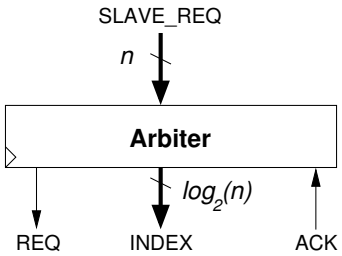


Figure 5.9: Arbiter component schematics.

[11] `http://opencores.org/project,gh_vhdl_library`

Round-Robin Arbiter. The framework provides a generic arbitration unit with the default implementation using the round-robin algorithm with time complexity $\mathcal{O}(n)$, where n is the number of possible requests coming at once. The component only selects the winner of the arbitration. Other logic (e. g. acknowledgement of the requesting components) is to be solved separately because it can diverse for different tasks.

Request-Acknowledger. The component contains logic that is required when serving requests on a bus. As such it can be used as acknowledgement generator for the arbiter component. The framework uses the component when dealing with AXI Lite Endpoint (see the figure 5.7). The default implementation provides a register level that delays the acknowledgement by one clock to improve timing. The number of requests can be quite high when serving a large address space with a lot of registers and that leads to long combinatorial paths.

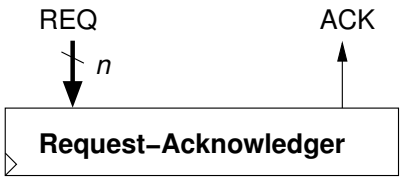


Figure 5.10: Request-Acknowledger component schematics.

Change Detector. This component is intended to be used for interrupt generation. It monitors a bit vector (`SIG`) and while its value differs from a predefined default value (`IDLE`) the component generates a high value—an interrupt to a processor (`EVENT`).

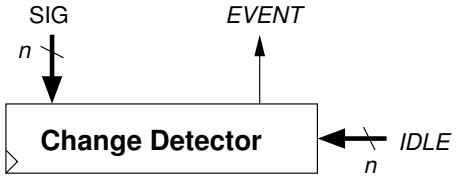


Figure 5.11: Change Detector component schematics.

Address Rebase. In some cases it is desirable to change a base of an address coming over an address bus. This component handles this task by inserting an adder into the address data path. Both the original and new base address must be specified before synthesis.

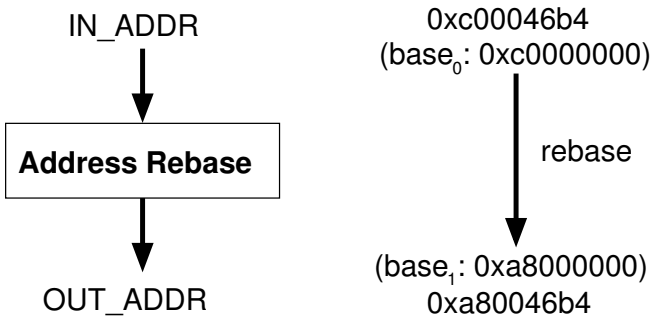


Figure 5.12: Address Rebase component schematics.

5.3.4 Function packages

The framework contains four function packages:

- `misc_pkg`—with general purpose functions and data type definitions (`log2`, `max`, `min`, `to_string`, `and_vector`, `or_vector`, `apply_be`[12], and others).
- `plat_pkg`—platform dependent functions and definitions (e. g. definition of `address` data type),
- `axi_pkg`—AXI related definitions,
- `rsoc_pkg`—RSoC Framework specific definitions.

The most notable is the `plat_pkg`. It defines data types that can depend on target platform. The most important data type is `address`. The default implementation (for Zynq) provides 32-bit addressing. The package defines function `compute_next_base` used to compute address space layout of the RSoC Bridge component. Its implementation is related to the AXI 1-to-N component, it must be aware of possibilities and limitations of the AXI 1-to-N architecture (e. g. granularity of addressing).

5.4 Infrastructure

The main part of the infrastructure is the RSoC Bridge as it was described in *5.2 RSoC Framework Overview*. Its main purpose is to provide bus systems connecting accelerators and the selected Processing System interfaces. The implementation delivered with the framework—*RSoC Bridge Generic*—has a unified structure and it is implemented in pure VHDL.

RSoC Bridge Generic. The component consists of several parts that are used as generators of lower level logic. The bus infrastructure is generated by the *Slave Bus* generator and *Master Bus* generator as described in the following text. Each accelerator has an associated controller that ensures the transaction delivery to and from the software part. The current implementation provides two types of controllers: *FIFO Interface* and

[12] Apply Byte-Enable when writing into a register.

Simple DMA Interface. Controllers are generated in the *FIFO IF Array* and *SDMA IF Array* structures. Some controllers are able to generate interrupts. The interrupt lines are connected out through the *Interrupt Mapper* as specified by its configuration. In some cases the AXI lines need to be reordered (remapped) for the implementation purposes to assure the correct interconnections. This task is done by the Remap components as it was described earlier in the text.

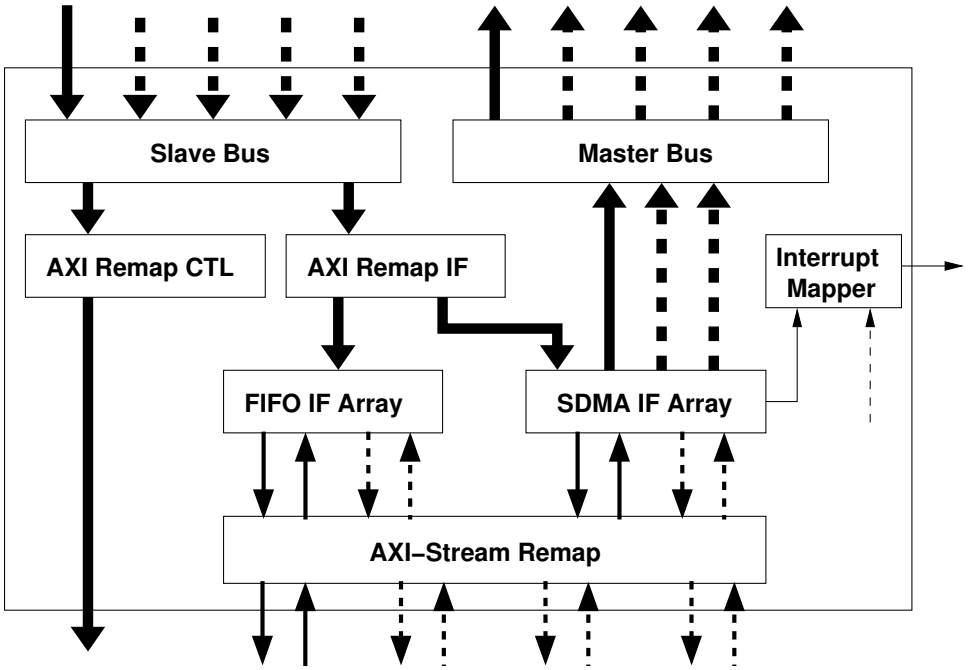


Figure 5.13: RSoC Bridge Generic architecture (dashed lines mark generic parts).

5.4.1 Slave Bus

The infrastructure needs a bus system to configure its internal units and to configure the accelerators. Such a bus system is generated by the Slave Bus. It generates an array of AXI 1-to-N components to connect the external masters to either accelerators and accelerators' controllers. The core generates one instance of the RSoC Info unit that delivers information about the address space layout to the operating system. The generator builds the address space layout during synthesis based on the information about address space of each particular accelerator and its controller.

5.4.2 Master Bus

The *Master Bus* connects internal bus master units, and the accelerators' controllers, to the Processing System. It generates an array of AXI N-to-1 components. The architecture enables sharing of data channels among many accelerators' controllers. This approach is

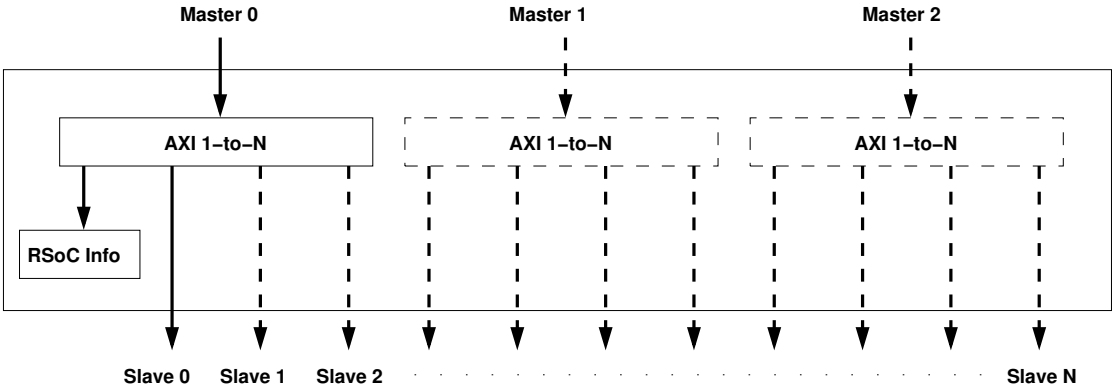


Figure 5.14: Slave Bus generator internals (dashed lines mark generic parts).

useful on platforms where there is very limited number of such channels between the PL and the PS. In general, it makes possible to include more accelerators in the design then the number of physical PS to PL channels.

5.5 Integration

The third goal of the framework is integration of the Processing System and the Programmable Logic. For that purpose there are hardware components intended for communication with software drivers. These pairs allows the complexity of communication between an accelerator and a software application to be hidden. There are two types of controllers supported, however, other two types are designed for the future extension of the framework.

5.5.1 Interface components (controllers)

Each controller is expected to provide an address space accessible from the Processing System. The first 32 B is reserved for information about the associated accelerator. The address space starting at offset 0x20 is controller specific.

FIFO Interface. The controller brings a simple way to communicate with an RSoC Accelerator in the design. Its main purpose is to save resources of the FPGA and to provide an easy way to debug an accelerator. Each data transaction is generated by accessing registers of the controller. This leads to very slow throughput, however, it is not the goal. The secondary purpose is to serve as a low-latency interface on platforms that has a support for it. The low-latency tasks are usually not data extensive and thus the throughput should not be an issue as well. It can be useful for accelerators with very short inputs and long outputs (or vice-versa) where a combination of the FIFO Interface and a DMA Interface can be used to handle the corresponding data streams.

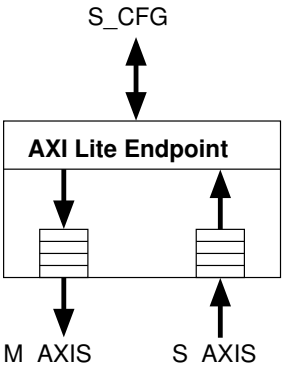


Figure 5.15: FIFO Interface component schematics.

The controller's address space consists of four registers: `STATUS`, `DATA`, `KEEP`, and `USER`. There is an obvious correspondence with the AXI-Stream simplex bus as described in *2.3.4 AXI4-Stream protocol*.

Simple DMA Interface. The second currently supported controller provides the Direct Memory Access capabilities. The controller exposes the register `STATUS` and four sets of other registers to the software driver. The controller manages DMA operations in both directions: memory-device and device-memory. For each direction there are two register sets called *request* and *response*. The former one is used to request a DMA transaction in the particular direction and the latter one returns status information about finished transactions.

When moving data into an accelerator, the s-request registers are used to setup the transaction. The driver must set the triple (address, size, id) to the corresponding registers `REQ_SADDR`, `REQ_SSIZE`, and `REQ_SID`. After the transaction is finished the s-response registers provide a pair (status, id) in the registers `RES_SSTATUS` and `RES_SID`.

For the opposite direction, the d-request registers are used to setup the transaction. The driver must set the triple (address, size, id) to the corresponding registers `REQ_DADDR`, `REQ_DSIZE`, and `REQ_DID`. After the transaction is finished the d-response registers provide a triple (status, size, id) in the registers `RES_DSTATUS`, `RES_DSIZE`, and `RES_DID`.

The `RES_xSTATUS` vectors provide information about the result of a transaction. The `RES_xSTATUS`$(1:0)$ bits match semantics of the `BRESP`/`RRESP` signal of the AMBA AXI protocol. The bit `RES_DSTATUS`(16) is set when a transaction device-memory has been truncated and therefore, the result is incomplete. Other bits are reserved for future use.

Each transaction is identified by an ID. It is an arbitrary number the driver selects (the process ID is suitable for this purpose). The ID can be used to match the related request and response. However, in the current implementation both the requests and responses are strongly ordered using a queue and thus the ID is of less importance.

The Simple DMA controller has several disadvantages. First, the CPU must configure each particular transaction which brings an unnecessary overhead into the communication process. The second notable problem is a necessity of using contiguous memory blocks.

There is no way how to specify more memory addresses in one request. However, the purpose of this controller is to save resources on the chip while providing fast processing of short data chunks up to about 8 KB. The memory limit of one transaction is given by the state of the system. If it is possible to allocate a long contiguous memory block then it is possible to use the controller for longer transactions as well. The third problem this controller brings is that the software driver does not know the size of receiving data chunk in advance. This can lead to situations when a data chunk is truncated to fill the allocated memory block. When a data chunk is truncated during a DMA transaction the 16th bit of the `RES_DSTATUS` register is set to one.

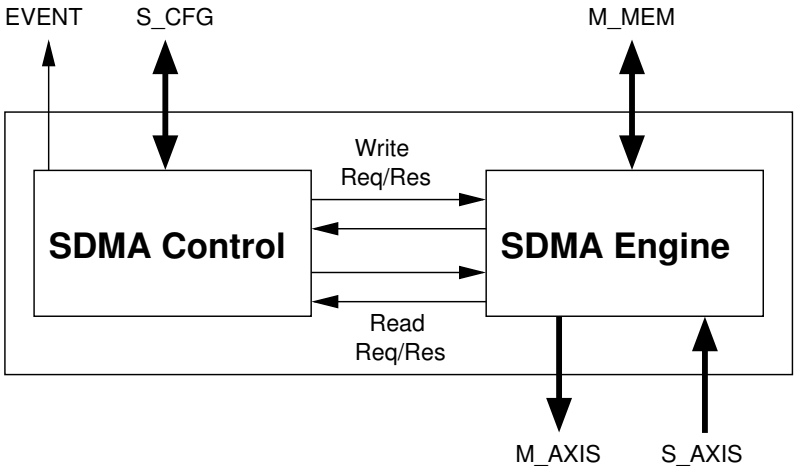


Figure 5.16: Simple DMA Interface component schematics.

The figure 5.16 shows the implementation of the SDMA Interface controller. It is divided internally into two sub components. The SDMA Control handles the address space access from the software and translates it into request and response streams connected into an SDMA Engine. The SDMA Engine is platform specific. The implementation for Xilinx Zynq is done using the AXI DataMover unit [24]. This approach makes the transaction engine independent on the software interface and that enables to have only one software driver for every SDMA Interface implementation.

Scatter-Gather DMA Interface. To solve the disadvantages of the Simple DMA Interface an advanced DMA unit can be used. It is known as a Scatter-Gather DMA. Such a device is given a list of descriptors that provide information about available memory chunks. The descriptors are initialized by the software driver and the address of the first one is written into the DMA controller. The controller then autonomously manages the descriptors and moves data between memory and a connected device. When the device starts to produce data the DMA engine selects the available memory blocks, writes the data there, and marks those blocks as used. The software driver is notified by an interrupt to check the descriptors and process the prepared data. When the data is processed the driver marks those blocks as free for reuse by the DMA system.

Such a device is to be integrated into the RSoC Framework to provide more powerful DMA to the accelerators. The designer would have an option to select which DMA

engine is better for a particular task. The Scatter-Gather DMA engine is more complex and consumes a lot more area on the chip. The second problem is that various engines of this kind can use different format of memory descriptors and that makes it nearly impossible to have a single software driver to handle them all.

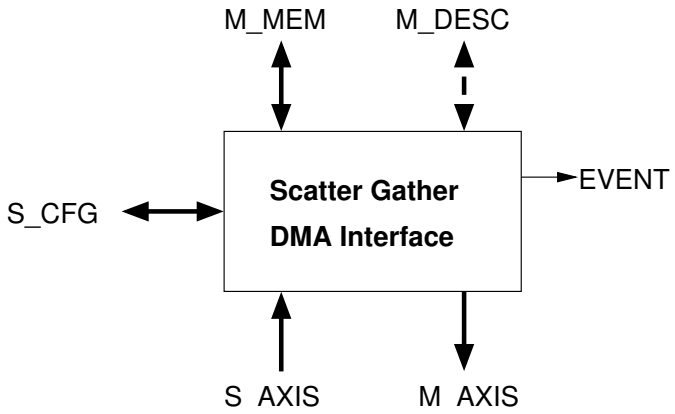


Figure 5.17: Scatter Gather DMA Interface component schematics.

For Xilinx platforms, the Xilinx AXI DMA IP core [25] can be used as the main DMA engine. The figure 5.18 shows the format of a buffer descriptor used by the IP core. Each field is 4 B long:

- **NXTDESC** is a pointer to the next descriptor,
- 4 bytes are reserved for future extension of the pointer to 64 bits,
- **BUFFADDR** represents address of the associated buffer,
- 12 B are reserved for future extensions, it can extend the buffer address to 64 bits if needed,
- **CONTROL** and **STATUS** holds information about the buffer size, frame boundaries and provides error reporting,
- **APP0–4** are 5 application-specific fields.

NXTDESC	4 B	BUFFADDR	12 B	CONTROL	STATUS	APP0–4

Figure 5.18: Descriptor used by the Xilinx AXI DMA IP core.

The IP core provides all the five interfaces shown in the figure 5.17 and adds three more: the control stream to start a DMA transaction (can be triggered e. g. when a frame comes from an accelerator) and two status streams, the first informs about finished device-to-memory transactions and the other informs about finished memory-to-device transactions. To implement the Scatter-Gather DMA engine for Zynq only some glue logic is needed: extraction of the frame size from the incoming data stream (from accelerator), insertion of the frame size into the outgoing data stream (into the accelerator), and proper handling of control and status streams the engine uses.

Central DMA Interface. Some platforms, and Zynq is the case, can provide a DMA engine (hardwired in the Processing System part or as a soft core, see [23]) that provides general DMA transactions among different slave components. Such a unit can be used to save resources in the Programmable Logic. If a hard IP of this kind is available, the saved in the logic can be significant. But also a soft central DMA can save logic because only one such engine is needed. This would influence the throughput of such components. The RSoC Framework can take advantage of it by implementing another type of interface controller.

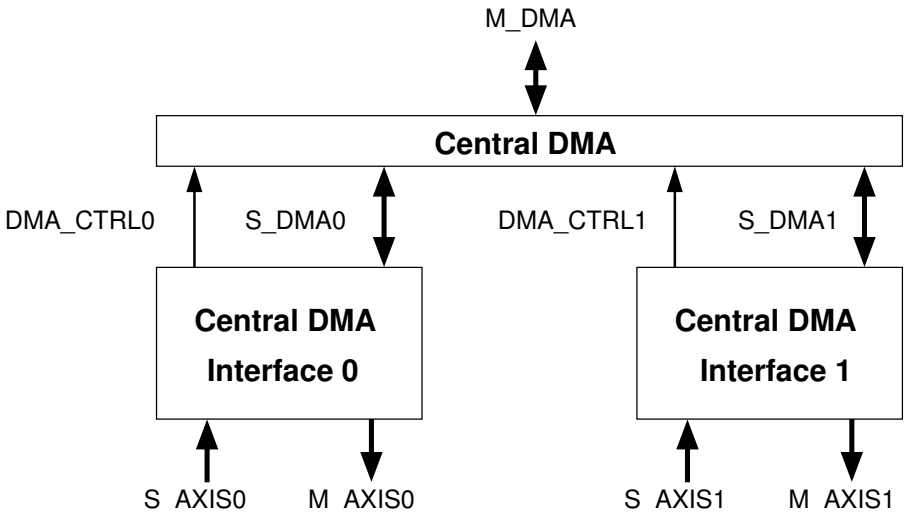


Figure 5.19: Central DMA Interface component schematics.

The figure 5.19 shows two such interface controllers connected to one central DMA engine. The engine can be either part of the Processing System or of the Programmable Logic as a soft core. In the second case, the port *M_DMA* would be connected to a slave AXI interface. In Zynq such a slave interface could be a High-Performance port or ACP. It is obvious that this approach opens another interesting feature. The accelerators connected through such a system can communicate to each other. At the moment, the accelerators lack an addressing mechanism to enable such a feature. However, the AXI-Stream protocol defines signal `TDEST` that may be utilized for this purpose.

The Central DMA Interface must be addressable by the Central DMA. The DMA transfers can be issued using the fixed burst transfers to map a memory address range to a single address of the controller. The Central DMA Interface then maps each AXI4 burst to the corresponding AXI-Stream connected to an accelerator.

5.5.2 Runtime components discovery

The software driver must be able to discover all accelerators and their associated controllers when probing. For this purpose a component RSoC Info has been implemented.

The component's address space contains descriptors of memory regions accessible via the RSoC Bridge. The RSoC Info component's address space starts by four registers:

- *NEG*—is read-write register that negates the written value. The software driver can use the register to verify the RSoC Info component at the given address is alive.
- *VERSION*—contains version of the RSoC Framework, i. e. 0x00000001 for the current version, where the lower two bytes (0x0001) represents the minor version number and the higer two bytes (0x0000) represents the major version number. The version of the current RSoC Framework is 0.1.
- *REGIONS*—contains the number of available regions.
- *REGION_OFF*—is an offset to the first descriptor from the beginning of the RSoC Info component's address space. This enables future extensions of the address space with backward compatibility.

Each region descriptor consists of three 4 B long entries and is padded to be 16 B long:

- *BASE*(i)—the absolute base address of the i-th memory region,
- *SIZE*(i)—the size of the i-th memory region, and
- *INFO*(i)—metadata describing what can be find in the i-th region.

Figure 5.20: Address space of the RSoC Info component.

Each such descriptor defines one addressable component in the system. The first regions describe the accelerators' address spaces and the following ones provides metadata about the controllers. The number of accelerators (and number of controllers) N can be counted as

$$N = \frac{\text{REGIONS}}{2}.$$

Every i-th accelerator corresponds to the $(i + N)$-th controller.

The RSoC Info component is automatically generated inside the Slave Bus of the RSoC Bridge Generic. In the Zynq platform, the component can be connected any of the two Master General Purpose ports. The base address to access RSoC Info is the address assigned to the RSoC Bridge on the configured AXI port. There is always exactly one such unit per bridge.

5.5.3 Generic software drivers

A software driver is needed to integrate the hardware soft components connected to the RSoC Bridge. Its purpose is to detect the RSoC Bridge from the device tree provided during the boot of Linux. The device tree provides the base address of the RSoC Info component and enables the driver to initialize the right drivers for each accelerator and its controller.

The first version of the driver is represented by a *platform driver*. The driver walks through the regions gathered from the RSoC Info component. For each known region it instantiates `struct rsoc_if`. Such a structure represents any of the supported controllers. The instance is initialized by a setup routine specific to the corresponding controller. The driver passes a preallocated device identification (*major* and *minor*).

Driver sdma-if. The driver handles one SDMA Interface controller. A write setups a new DMA transaction into the controller and a read setups a new DMA transaction from the controller.

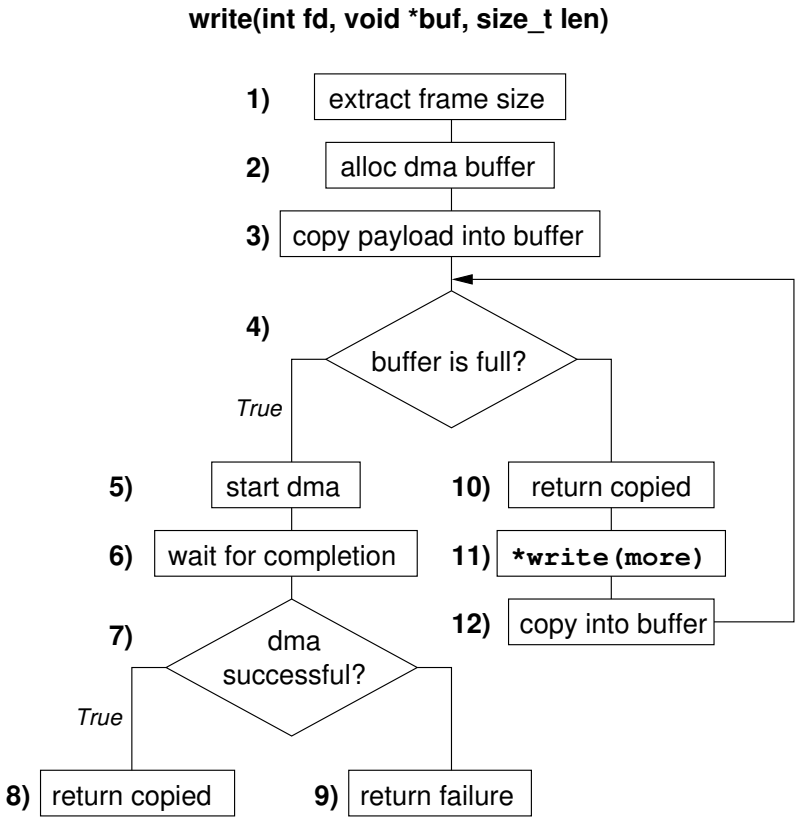


Figure 5.21: Simplified algorithm performed by each SDMA `write()` operation.

The figure 5.21 summarizes the steps of the implementation of SDMA `write()` operation. The steps are discribed in detail:

1. The frame size is extracted from the first four bytes of the passed user data buffer. If the caller passes less then four bytes in the first call to `write()`, the driver returns `-EINVAL` error code.
2. A contiguous memory block is preallocated to fit the frame that is to be written. The block is managed inside an instance of `struct rsoc_buf`. The structure simplifies working with the buffer and associates the buffer to the callers Process ID (PID). The following writes by the same process would continue to fill the buffer until it is full.

3. The frame can be transferred when all its contents were passed to the driver, i. e. when the buffer is full.
4. The DMA is started. This comprises mapping of the buffer by the Linux DMA API and writing the registers of the corresponding SDMA Interface controller. The PID is used as the ID of the transfer.
5. The driver now waits for completion of the DMA transfer. A kind of semaphore is used for this purpose. When the DMA transfer is done an interrupt is issued and handled by the driver. The interrupt routine reads the result of the transfer. Then it updates the buffer indentified by ID of the result. Finally it wakes up the process blocked by waiting for completion.
6. After the driver is awaken, it checks the result of the DMA transaction.
7. If the transaction was successful, it returns the number of bytes written by the last `write()` call.
8. If the transaction was not successful, the driver returns `-EIO` error code.
9. If the buffer is not full, no DMA transaction is started yet. Instead the driver returns to the process expecting that more data will come later. It returns the number of bytes passed with last `write()` call.
10. The process writes another part of the frame.
11. The driver copies the given data into the buffer and tries to start the DMA transfer if the buffer is full.

The figure 5.22 summarizes the steps of the implementation of SDMA `read()` operation. The steps are discribed in detail:

1. The frame size *hint* is extracted from the first four bytes of the passed user data buffer. The driver expects the size hint to preallocate buffer of enough size. If the caller passes less then four bytes in the first call to `read()`, the driver returns `-EINVAL` error code.
2. The driver preallocates a buffer for the incoming DMA transaction of the size specified by the hint. The buffer is managed inside an instance of `struct rsoc_buf`. The instance associates the buffer to the callers Process ID (PID). The following reads by the same process would continue to read from the buffer until it is empty.
3. The DMA is started. This comprises mapping of the buffer by the Linux DMA API and writing the registers of the corresponding SDMA Interface controller. The current PID is used as the ID of the transfer.
4. The driver waits for completion of the DMA transfer. A kind of semaphore is used for this purpose. When the DMA transfer is done an interrupt is issued and handled by the driver. The interrupt routine reads the result of the transfer and updates the corresponding buffer. Finally it wakes up the process blocked by the waiting for completion.
5. The driver is awaken and checks the result of the transaction.

6. The transaction was successful. It writes the size of the transfer into the first four bytes of the user buffer.
7. The driver copies as much data as possible from the DMA buffer into the user buffer.
8. The DMA buffer is checked for available data.
9. If the DMA buffer is empty, the driver returns to the user process and passes the size of copied data. If it returns for the first time, it must add 4 for the frame header.
10. If the buffer is not empty the driver returns the number of copied data (plus 4 for the first return). It expects the process to call the `read()` again.
11. The process calls another `read()` to retreive the rest of the buffer.
12. If the DMA was not successful, the error code `-EIO` is returned.

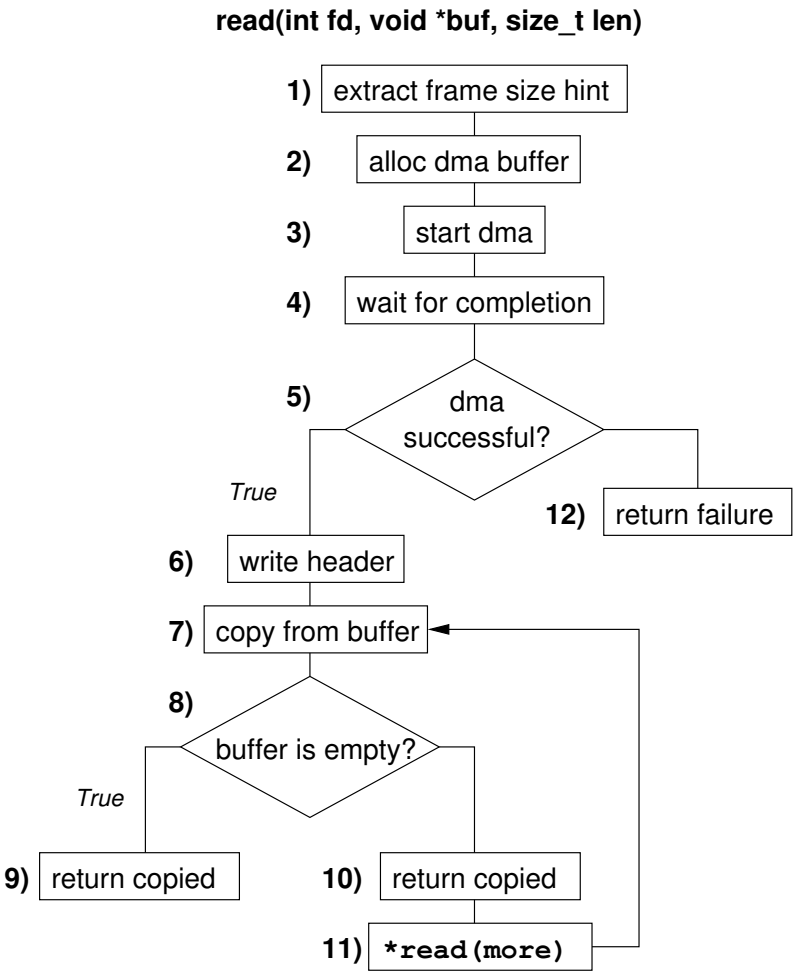


Figure 5.22: Simplified algorithm performed by each SDMA `read()` operation.

To simplify the implementation of both the read and write algorithms, each instance of `struct rsoc_buf` has an associated Process ID (only one read and one write buffer is permitted per process) and a state machine as shown in the figure 5.23.

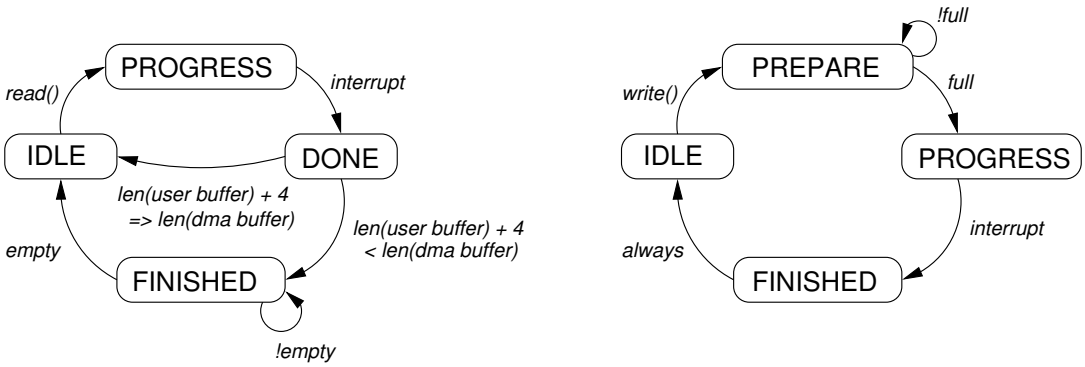


Figure 5.23: The state machine used by the read buffer (left) and write buffer (right).

5.6 Portability

The portability of the RSoC Framework hardware part can be seen at those levels:

1. the implementation language portability,
2. the component implementation portability, and
3. the system architecture portability.

5.6.1 HDL language

The language used to implement the RSoC Framework hardware components is VHDL. The VHDL is a language with rich set of features and constructs but only a subset of the language can be used to design hardware components. This fact is well-known and accepted by the hardware designers. This leads to another issue: some constructs are not well supported by the synthesis tools and when they are used for synthesis-time computations (not for the real synthesis) the tools can fail to proceed.

The VHDL has several limitations that are usually solved by using an external language (TCL, Python, etc.). This makes the whole system less portable and for that reason it was refused to be used for component and infrastructure parts of the RSoC Framework. As a result any synthesis or simulation system supporting VHDL at the neccessary level can be used to compile and integrate the RSoC Framework.

Using only the basic VHDL leads to pure code that is difficult to read and review. For a system equipped with complex interconnections this can be seen as an important problem. As the implementation advances the system is more difficult to extend and at some point it starts to be unmaintainable. The RSoC Bridge Generic component is a complex system that is difficult to describe using just the basic VHDL constructs. It uses several “not so frequently used” constructs that still pass the synthesis tools provided by Xilinx. Other synthesis tools were not tested yet.

5.6.2 System architecture portability

In the context of portability the greatest value of using the RSoC Framework is that an already working application on one RSoC architecture can be migrated to another one

without any or few changes in the application code. The configuration of the framework is hidden from the application specific parts and the impact of a migration can be seen more on the behavioral level. The application moved to a different chip can be slower because the new chip provides lower throughput between the PS and the PL, or, more likely, faster because of a more powerful interconnections are available. Another important parameter of an architecture can be the possibility of real-time processing where guaranteed low-latency communication channels are required. The RSoC Framework does not differentiate between low-latency or high-speed channels and so an application can be easily accomodated and improved by a migration.

The figure 5.24 shows a migration of an application, composed of accelerated software services, to another platform. The only important changes can be found on the boundary of the PS and PL (consider the same OS) and those changes are completely under control of the RSoC Framework. The second platform (on the right) provides only one high-speed channel that impacts the throughput among services and accelerators.

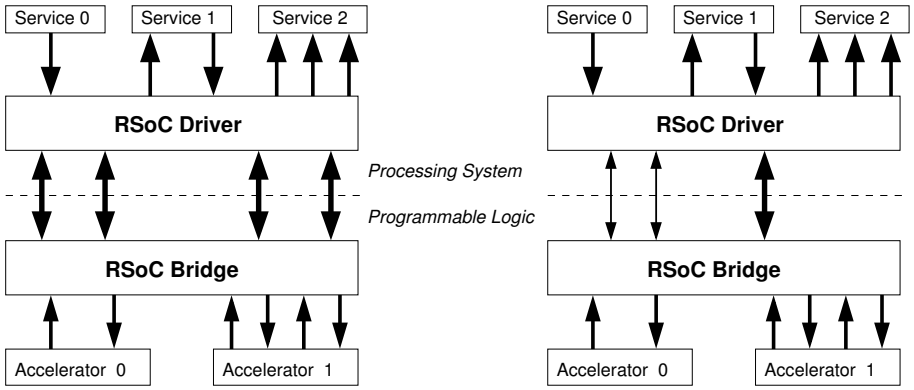


Figure 5.24: Migration from one architecture to another.

6 Designing with RSoC Bridge

In this chapter a few examples of using the RSoC Framework introduced in this work are shown.

6.1 Generic example

For testing purposes, the RSoC Framework contains the so called *Loopback Accelerator* unit. It emulates a working accelerator in the Programmable Logic. It copies data from the input to the output port. It provides two registers over the configuration port to read how many frames and data beats were seen by the unit.

The testing architecture is shown in the figure 6.1. It represents a general use case for such a system. The RSoC Bridge is connected to the Zynq Processing System (using a wrapper Processing System 7 by Xilinx) and provides connections for four Loopback Accelerators.

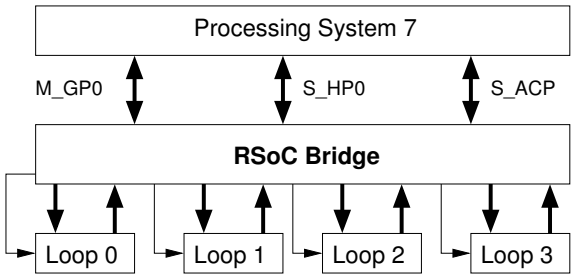


Figure 6.1: Generic architecture used for testing purposes.

The accelerators Loop 0 and Loop 1 are connected using the FIFO Interface, the Loop 2 is connected using SDMA Interface via the port HP 0, and the Loop 3 is connected using SDMA Interface via the port ACP.

The architecture was synthesized for chip *xc7z020-1 clg484* (i. e. Zynq 7020). It consumes 7,220 LUTs (13 % of the chip) and 7,271 Flip-Flip registers (6 % of the chip) while the RSoC Bridge itself is estimated[13] by the XST compiler to be 8,407 LUTs and 6,853 Flip-Flops in this configuration. The table 6.1 shows estimated resources of the selected components used inside the RSoC Bridge.

The measured throughput of the SDMA Interface with non-coherent access is 20 MB/s (measured with 1 MB long frames) and decreases with the growing size of frame. It was not possible to allocate kernel buffers for frames of size greater then 5 MB. The throughput is

[13] Phases following the synthesis (Map and Place & Route) shrink the consumed resources.

Component	LUTs	Flip-Flops	Frequency	Notes
sdma-if	2,821	1,588	287 MHz	
fifo-if	418	182	465 MHz	
axi-lite-endpoint	384	117	364 MHz	
axi-1ton	213	122	373 MHz	$n = 4$
axi-nto1	211	218	538 MHz	$n = 4$

Table 6.1: Resources of components provided by the RSoC Framework

limited because the driver copies data from userspace into the kernel buffer without using the DMA. A zero-copy approach, using the memory mapping capabilities of the kernel, can improve the throughput by avoiding the unnecessary copies.

6.2 Dynamic reconfigurable accelerator

The RSoC Framework is designed with dynamic partial reconfiguration in mind (however, it is not supported yet). In the future versions, an accelerator could be a loadable entity at runtime.

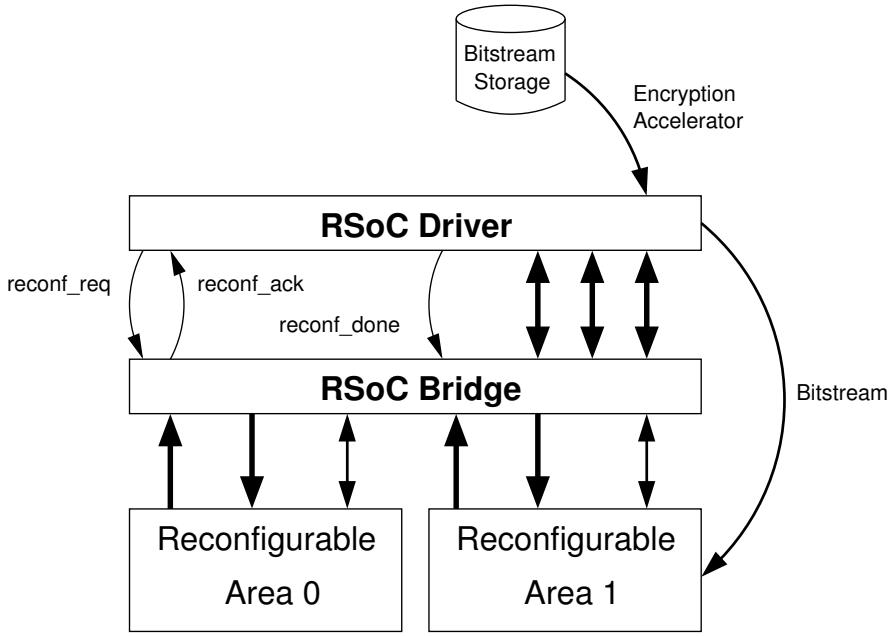


Figure 6.2: Partial Dynamic Reconfiguration with RSoC Bridge.

Consider an application with variable load. If there is no or a little work to do, all the computations can be done by the Processing System only. The PS itself can take advantage of CPU sleep states to save energy. However, when the load grows up the system can detect it and select a hardware module to be downloaded into a prefined area

in the PL. The software application will be able to detect a new hardware accelerator and start using it. This increases the power consumption, however, the system is able to service the great amount of requests coming into the system. When the load drops the accelerator can be removed from the PL and the system can run low power again.

To enable such a scenario, the framework must be notified that a partial reconfiguration is about to occur (`reconf_req`, `reconf_ack`) and prepare the bus systems for it (e. g. pause all transactions, this behaviour is platform dependent). After the reconfiguration finishes (`reconf_done`) a new accelerator can be detected and registered by the operating system. For the OS, this operation may look like hot-plug of a USB stick. The new accelerator is recognized by its information vector.

7 Conclusion

The Xilinx Zynq has been described with respect to the available interfaces between the Processing System (PS)—ARM cores—and the Programmable Logic (PL)—FPGA—on the chip. The interfaces are important part when considering the HW/SW codesign methodology. There are two types of high-speed channels available to the Programmable Logic that allows to access the main memory of the system (the memory is shared by both the PS and PL). One of those channels is the Accelerator Coherency Port that ensures coherency among the L1 and the L2 caches. This can simplify and speed up the software when dealing with hardware because it does not need to flush the caches for DMA transfers.

Developing applications for such a platform, while considering both hardware and software design, requires knowledge about both the hardware technologies and operating system's (Linux in this case) internals. When accelerating a software part of an application in hardware, two components are always required: a *software driver* (especially for systems with an operating system) and a *hardware controller* that performs DMA transfers. It is possible to have a generic driver and a corresponding hardware controller to integrate various types of accelerators or adapters into the system. This simplifies the development and improves the time-to-market factor. However, when developing more complex hardware support, the complexity of interconnections inside such a system increases together with the required resources. The performance of the system becomes less predictable. Therefore, various types of DMA controllers with different characteristics may be needed. The easier is to change one controller for another the more configurations can be tested to find the best one.

In this work the RSoC Framework suitable for HW/SW codesign has been introduced and prototyped. It covers three areas:

1. It provides *reusable hardware components* to accelerate the hardware development and support portability of the system.
2. The RSoC Bridge component provides generation of *infrastructure*, an internal bus system that connects the user hardware accelerators to the Processing System via the selected controllers with only limited effort of the developer.
3. The controllers together with generic software drivers enables *integration* of the hardware system into an application (a new one or an existing one).

The Xilinx Zynq is not the only device consisting of a Processing System and a Programmable Logic and so a more general group of devices have been covered. The Reconfigurable System-on-Chip (RSoC) platforms extends the well-known class of devices denoted as System-on-Chip. Because the RSoC platforms group is growing rapidly the concepts described in this work are generalized to be applicable to various such systems. Once an application is developed using the RSoC Framework, it is possible to migrate to

another chip with just minor or no changes to the whole application because the specifics of the platform are hidden by this framework.

The framework is intended to support the partial dynamic reconfiguration of the connected accelerators in the future. This can open new possibilities to HW/SW codesign of applications. Dividing computations between hardware and software can be done at runtime on demand. This can improve power consumption and reduce the required area on the chip while providing the high throughput of the system.

Bibliography

[1] Corbet, J.—Rubini, A.—Kroah-Hartman, G. *Linux Device Drivers*. Third edition. O'Reilly Media, February 2005. URL: http://lwn.net/Kernel/LDD3/. ISBN: 0-596-00590-3.

[2] Love, R. *Linux Kernel Development*. Third edition. Pearson Education: 2010. ISBN: 978-0-672-32946-3.

[3] Noergaard, T. *Embedded Systems Architecture*. Elsevier, 2005. ISBN: 0-7506-7792-9.

[4] Platzner, Marco at al. *Dynamically Reconfigurable Systems*. Springer, 2010. ISBN: 978-90-481-3484-7.

[5] SLABÝ, Jiří. *Rapid Data Transfers on COMBO Platform*. 2008. Thesis. Masaryk university, Faculty of Informatics. Advisor Pavel Čeleda.
URL: http://is.muni.cz/th/98734/fi_m/.

[6] Wolf, W. *Multiprocessor Systems-on-Chips*. Elsevier, 2005. ISBN: 0-12385-251-X.

[7] Altera. *Arria V Device Handbook: Hard Processor System Technical Reference Manual* [online]. Volume 3. November 2012. Chapter 5, HPS-FPGA AXI Bridges. URL: http://www.altera.com/literature/hb/arria-v/av_54005.pdf (May 2013).

[8] ARM. *AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI-4Lite, ACE and ACE-Lite* [online]. 2011. 306 p.
URL: https://silver.arm.com/download/download.tm?pv=1198016 (May 2013).

[9] ARM. *AMBA 4 AXI4-Stream Protocol* [online]. 2010. 42 p.
URL: https://silver.arm.com/download/download.tm?pv=1074010 (May 2013).

[10] ARM. *Application Note 228: Implementing DMA on ARM SMP Systems* [online]. August 2009. 15 p. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dai0228a/DAI228A_DMA_on_SMP_systems.pdf (May 2013).

[11] ARM. *ARM Architecture Reference Manual: ARMv7A and ARM-v7R edition* [online]. December 2011. 2158 p. URL: https://silver.arm.com/download/download.tm?pv=1299246 (May 2013).

[12] ARM. *Cortex-A Series: Programmer's Guide* [online]. June 2012. 451 p.
URL: https://silver.arm.com/download/download.tm?pv=1296010 (May 2013).

[13] ARM. *Cortex-A9 MPCore: Technical Reference Manual* [online]. June 2012. 124 p.
URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407i/DDI0407I_cortex_a9_mpcore_r4p1_trm.pdf (May 2013).

[14] Corbet, J. *Supporting multi-platform ARM kernels* [online]. LWN.net. May 2012.
URL: http://lwn.net/Articles/496400/ (May 2013).

[15] Corbet, J. *The cdev interface* [online]. LWN.net. August 2006. URL: http://lwn.net/Articles/195805/ (May 2013).

[16] Likely, Grant. *Linux and the Device Tree* [online]. November 2012. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/devicetree/usage-model.txt (May 2013).

[17] Miller, D.—Henderson, R.—Jelinek, J. *Dynamic DMA mapping Guide* [online]. February 2013. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/DMA-API-HOWTO.txt (May 2013).

[18] Mitič, M.—Stojčev, M. *An Overview of On-Chip Buses* [online]. December 2006. URL: http://www.doiserbia.nb.rs/img/doi/0353-3670/2006/0353-367006034-05M.pdf (May 2013).

[19] Open Firmware Working Group. *Open Firmware Home Page* [online]. May 2005. URL: http://www.openfirmware.org/1275/home.html (May 2013).

[20] So, Hayden Kwok-Hay and Brodersen, Robert W. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers* [online]. EECS Department, University of California, Berkeley. Jul 2007. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html (May 2013). UCB/EECS-2007-92.

[21] Xilinx. *7 Series FPGAs Overview* [online]. 2012. 16 p. DS180. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (May 2013).

[22] Xilinx. *AXI Reference Guide* [online]. 2012. 132 p. UG761. URL: http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf (May 2013).

[23] Xilinx. *LogiCORE IP AXI Central Direct Memory Access (v3.00.a)* [online]. March 2011. DS792. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v3_03_a/pg034_axi_cdma.pdf (May 2013).

[24] Xilinx. *LogiCORE IP AXI DataMover v3.00a* [online]. October 2012. PG022. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v3_00_a/pg022_axi_datamover.pdf (May 2013).

[25] Xilinx. *LogiCORE IP AXI DMA (v3.00a)* [online]. March 2011. DS781. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v3_00_a/axi_dma_ds781.pdf (May 2013).

[26] Xilinx. *Virtex-5 Family Overview* [online]. Fabruary 2009. 13 p. DS100. URL: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf (May 2013).

[27] Xilinx. *XA Zynq-7000 All Programmable SoC Overview* [online]. 2012. 25 p. DS188. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (May 2013).

[28] Xilinx. *Zynq-7000 All Programmable SoC: Technical Reference Manual* [online]. 2012. 1707 p. UG585. URL: http://www.xilinx.com/support/documentation/user_guidesug/585-Zynq-7000-TRM.pdf (May 2013).

Appendix

The CD contains the following directories:

- `rsoc-framework`: sources of the RSoC Framework (both HW and SW) and testing application *testio*.
- `buildroot`: support for Zedboard, RSoC Framework and *testio* for the Buildroot.
- `test-design`: EDK project of the design specified in *6.1 Generic example*.
- `binary`: prebuild binaries of the design and the operating system.

The necesary steps to test the system:

1. Copy the files from `binary/` directory to the original Zedboard SD card (a working U-Boot is already there).
2. Power on the board and stop the autoboot.
3. Boot using the provided script:

   ```
   > mmcinfo; fatload mmc 0 0x4000000 uboot-startup.bin; source 0x4000000
   ```

4. Login as *default* and switch to *root* by calling `su`.
5. Execute

   ```
   $ mount /dev/mmcblk0p1 /mnt
   $ mknod /dev/xdevcfg c 259 0
   $ cat /mnt/system.bit.bin > /dev/xdevcfg
   $ insmod /lib/modules/3.8.0-xilinx/extra/rsoc_bridge_drv.ko
   $ mknod /dev/test0 c 248 0
   $ mknod /dev/test1 c 248 1
   $ mknod /dev/test2 c 248 2
   $ mknod /dev/test3 c 248 3
   $ dd if=/dev/urandom bs=32 count=1 | testio 32 1 /dev/test0 | wc -c
   $ dd if=/dev/urandom bs=32 count=1 | testio 32 1 /dev/test1 | wc -c
   $ dd if=/dev/urandom bs=32 count=1 | testio 32 1 /dev/test2 | wc -c
   $ dd if=/dev/urandom bs=32 count=1 | testio 32 1 /dev/test3 | wc -c
   ```

6. Each call to *testio* should print 32, the number of bytes sent through the system.