TECHNISCHE UNIVERSITÄT DARMSTADT

Embedded Systems & Applications

Hardware-acceleration of Java-implemented experiments on an open satellite.

Bachelorthesis by
Jan-Peter Ceglarek

November 2017

Examiner:
Prof. Dr.-Ing. Andreas Koch
Supervisor:
Dr.-Ing. Andreas Engel

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Hardware-acceleration of Java-implemented experiments on an open satellite.
Hardwarebeschleunigung von in Java implementierten Experimenten auf einem Open-Satellite.
Bachelorthesis by Jan-Peter Ceglarek
Submitted on 01.11.2017
Examiner: Prof. Dr.-Ing. Andreas Koch
Supervisor: Dr.-Ing. Andreas Engel

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, 1. November 2017

(Jan-Peter Ceglarek)

Abstract

The OPS-Sat project is the first small satellite mission launched by the European Space Agency (ESA). Its an open satellite using modern commercial hardware and with which procedures to change the onboard software during the flight will be tested. As its payload the satellite uses a MitySOM-5CSX - a System on Chip (SoC) device featuring an Intel Cyclone V Field-Programmable Gate Array (FPGA) fabric as well as an Advance RISC Machine (ARM) processor. The mission requires to run Java and C programs on the satellite.
In order to maximize the satellite's performance, the FPGA will work as a hardware-accelerator for generic Java code. Therefore particularly suitable parts of the Java code should be executed on the FPGA. Those segments could be selected by the developer with an Integrated Development Environment (IDE) plugin. Analysis, of which Java construct are suited to be hardware-accelerated on the FPGA, is however out of the scope of this thesis. This thesis demonstrates the necessary internal communication between hard- and software processor. Therefore two example implementation, using the Advanced eXtensible Interface lightweight bridge, based on the AXI light standard, were implemented. The first example only uses the FPGA to explain the basic development toolchain. The second one implements the actual communication bridge mastered by the HPS. A C program running on the ARM uses memory mapping to communicate with the interface.
An analysis of existing solutions shows, that there is no practicable tool to execute Java on an FPGA, or to convert Java to either C or Hardware Description Language code. But there are however working solutions to generate HDL code out of C code. In order to help developers with the conversion from Java to C, an Eclipse plugin can make annotations on selected lines in the code. These annotation can be used by others tools to generate generic C code automatically.

Kurzfassung

Das OPS-Sat Projekt ist die erste Kleinstsatelliten Mission, die von der Europäischen Weltraum Agentur (ESA) ins Leben gerufen wurde. Mit diesem Open-Satellite, der aus moderne konventionelle Hardware besteht, sollen Verfahren getestet werden, die Onboard Software während des Missionsbetriebes zu tauschen. Der Satellite nutzt als Payload einen MitySOM-5CSX - ein System on Chip (SoC) bestehend aus einen Intel Cyclone V Field-Programmable Gate Array (FPGA) und einem Advance RISC Machine (ARM) Prozessor. Die Missionsrichtlinien setzen voraus, dass sowohl Java, als auch C Code auf dem Satelliten ausgeführt werden kann.
Um die Leistungsfähigkeit des Satelliten zu erhöhen, soll der FPGA als Hardwarebeschleuniger für generischen Java Code genutzt werden. Dafür sollen geeignete Abschnitte des Java Codes auf dem FPGA ausgeführt werden. Diese Abschnitten können zuvor von dem Entwickler mittels eines Integrated Development Environment (IDE) plugins ausgewählt werden. Eine Analyse, welche Java Konstrukte besonders geeignet, sind um mit einem Hardwareprozessor beschleunigt zu werden, ist nicht Bestandteil dieser Arbeit. Mit dieser Thesis soll demonstriert werden, wie die benötigte interne Kommunikation zwischen dem Hardware- und dem Softwareprozessor realisiert werden kann. Dazu wurden zwei Beispiele implementiert, welches die Lightweight HPS-2-FPGA Bridge, ein SoC internes Interface basierend auf dem Advanced eXtensible Interface (AXI) light Standard, verwendet. Das erste Beispiel verwendet ausschließlich den FPGA, um die Entwicklung einer FPGA-Programmierung für die verwendete Hardware zu demonstrieren. Das zweite implementiert die eigentliche Kommunikationsbrücke, die von dem HPS gemastert wird. Ein auf dem ARM ausgeführtes Programm kommuniziert mit dem Interface mittels Memory Mapping.
Eine Analyse der bereits existierenden Lösungen Java Byte-Code auf einem FPGA auszuführen zeigt auf, dass es keinen brauchbares Verfahren gibt. Keines der vorgestellten Lösungen bietet eine, den Anforderung genügende, Übersetzung von Java nach HDL, oder kann Java direkt auf FPGA ausführen. Es gibt jedoch Programme, die HDL Code aus C Code generieren können.

Um den Entwicklern bei der Übersetzung von Java nach C zu helfen, soll ein IDE Plugin Annotations zu ausgewählten Zeilen im Java Code hinzufügen können. Diese Zusätze können dann in weiteren Schritten genutzt werden, um automatisiert

C Code zu generieren.

Contents

1 Introduction

The following chapter provides the reader an overview of the topics covered, and also with the motivation and definition of this current thesis. The last paragraph gives an outline of the following chapters and summarizes their content.

1.1 Motivation

Nano satellites, weighing around 10 kg, are way lighter and much less powerful than their bigger companions. Caused by their smaller size, their possibilities are strictly limited. The big advantage of those small satellites is, that they are relatively cheap. Manufactures sell custom nano satellites for a few hundred thousand Euros. The OPS-Sat project [18] is the first small satellite mission launched by the European Space Agency (ESA). Its main goal is to test methods to change the onboard software and to test normal commercial, instead of special space-tested hardware. For this reason companies or private makers are invited run experiments on this open satellite. The submitted experiments can be either written in C or Java, so the satellite must be able to handle both programming languages. In order to increase the performance of the software processor, as well as the data throughput of the whole satellite, the Field-Programmable Gate Array (FPGA) will be used as a hardware-accelerator for executing Java byte-code. Therefore the Java code needs to be converted to Hardware Description Language (HDL) and an analysis of the existing tools revealed, that none of them provides satisfying results. Furthermore it is particular new to this mission, that new software should be transferred to the payload during flight and therefore changing the programming without physically touching the FPGA is absolutely necessary. So the Advance RISC Machine (ARM) has to be able to reprogram the FPGA.

1.2 Scope of the Thesis

Since it is key to the mission to run Java code on the FPGA, existing conversion tools have to be analyzed and decided, whether they match the requirements.

There are different High-Level Systhesis (HLS) tools available, but none of them are working with Java code. All of them are translating C code to one of the two HDLs. So in order to work with those tools, general Java code has to be converted into C code. This thesis is also about getting an overview over the current situation on how it is possible to convert Java to C code, what the existing programs are and if there is a working solution.

In terms of performance, it is not recommended to perform Java code on an FPGA instead of an ARM, because every Java code must be interpreted or cross compiled before it can run on the device. Nevertheless it can be of advantage to run specific patterns on the FPGA. Therefore an Integrated Development Environment (IDE) plugin should help the developer to convert selected parts of the code into C code, so that a HLS tool can compile it into HDL later on. In order to use the FPGA for executing Java code, extraction and preparing of hardware-kernels out of general Java code is necessary. There are different approaches to solve this problem. Since there is no solution, which performs good enough on an FPGA or supports newer Java versions since 1.5, a refactoring-tool for a Java IDE should help developers to work with the hardware offered by the satellite. This tool will help the developers to generate C code out of general Java byte-code. Eclipse was chosen as the fitting IDE, because it is the most used IDE for Java development, it has a variety of plugins and is known for making it user-friendly to develop new plugins.

To program the FPGA from the ARM, internal communication channels are used. The options Cyclone V is offering the user will be discussed and how to implement one of them. Because exchanging the programming of the FPGA has to be done without touching the hardware, it is necessary to use the ARM processor for reprogramming the FPGA and not using an USB blaster or flashing the SD card manually. This thesis will cover different approaches to change the programming of the FPGA and how it is specifically done with the used MitySOM-5CSX. The analysis of the existing solutions will show, that none of these tools complies with the requirements. Hence it is also part of this thesis to start the development of an IDE/Eclipse plugin, that will help the developer with this conversion. The plugin should annotate lines in the code selected by the user with *@FPGA*. These annotations should be used later on by conversion tool.

1.3 Structure

This thesis is structured into six chapters. Beginning with the introduction in, the motivation as well as the purpose of this thesis is described. Chapter 2 will give the reader more informations about what the used MitySOM-5CSX is and how the communication between the integrated FPGA fabric and the Hard Processor System (HPS) works, what Memory Mapping is and why it is essential for the communication with an System on Chip (SoC). Further the Eclipse plugin is explained in more detail. This includes an explanation on how the basic structure of an Eclipse plugin works and what is needed to build a new one in Section 2.2. The third chapter will give an overview about existing conversion tools. What the differences are and why none of them are suited for the purpose of this work, is described in Section 3.4. In Chapter 4 it is described, how the implementation of the Advanced eXtensible Interface (AXI) communication bridge within the MitySOM-5CSX was done and how the FPGA can be programmed with and without using the software processor.

The results of the implementation are presented in Chapter 5.

2 Technical Background

Starting with a visualization of the implementation, the used hardware and tools are described in Section 2.1. Section 2.2 focusses on information about the plugin. The actual implementation is described in Chapter 4.

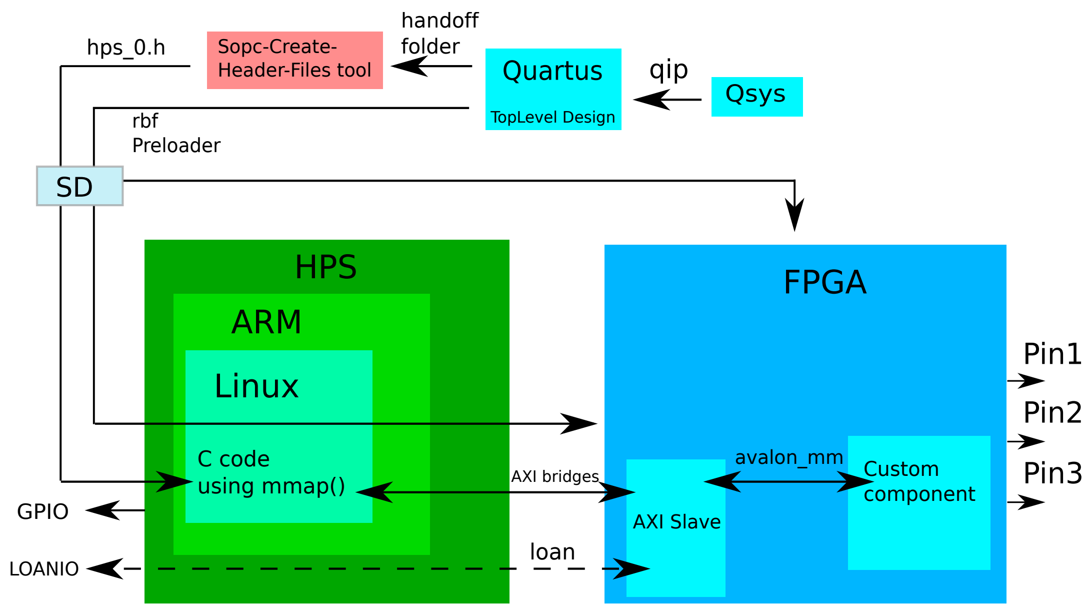


Figure 2.1: The concept of the whole tool chain

2.1 The MitySOM-5CSX used by the OPS-Sat project and its development tools

The following section describes the used hardware and the essential tools to configure the FPGA.

2.1.1 Hardware Specifications

This section focuses on the hardware used for this thesis.

The MitySOM-5CSX

The OPS-Sat project uses a MitySOM-5CSX System on Module (SOM) and its development kit build by CriticalLink [17]. The MitySOM-5CSX board features an Altera Cyclone V SoC [9], which consists of an FPGA and an ARM processor. Each of them has its own power supply, RAM, peripherals and pins. They can work together via their internal communication bridges or fully independent from each other.

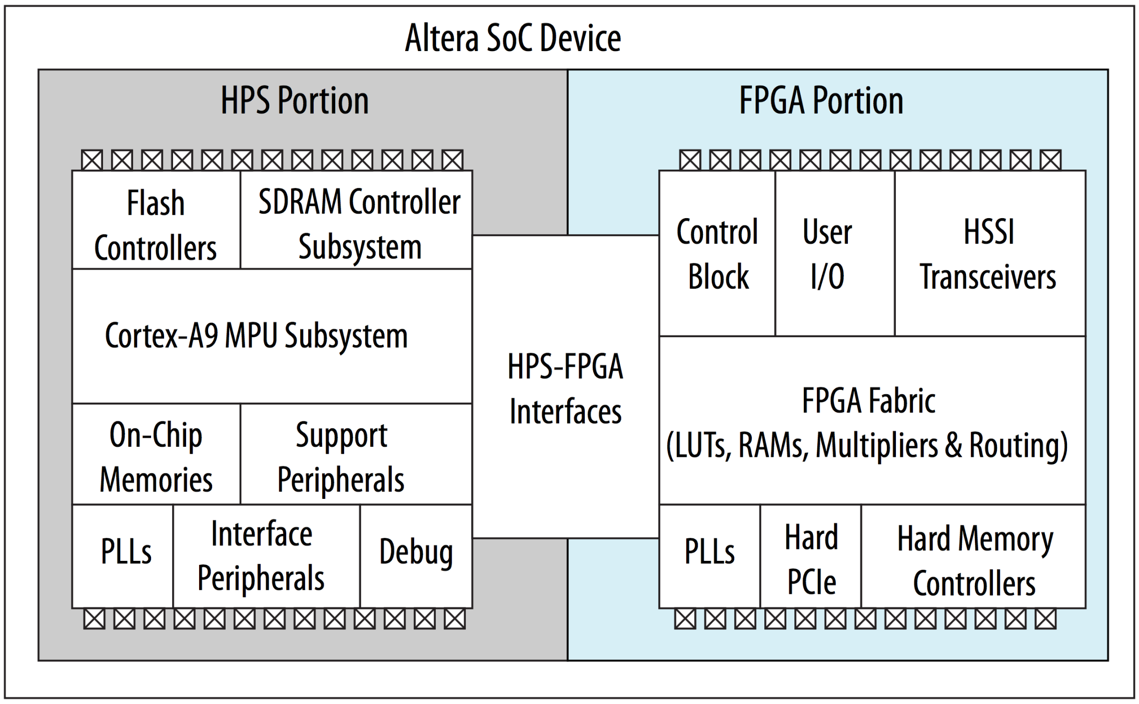


Figure 2.2: The SoC block diagram from the manual [5]

The Hard Processor System

The Hard Processor System (HPS) is one of the two elements of the Cyclone V. Inside the HPS, the ARM is the main part. Several peripherals and controllers are also part of the HPS. The single or dual-cored ARM Cortex, clocked with a maximum frequency of 925 MHz, can work fully independently, and has several connections to communicate with the FPGA as shown below. These bridges are described more detailed in Section 2.1.2.

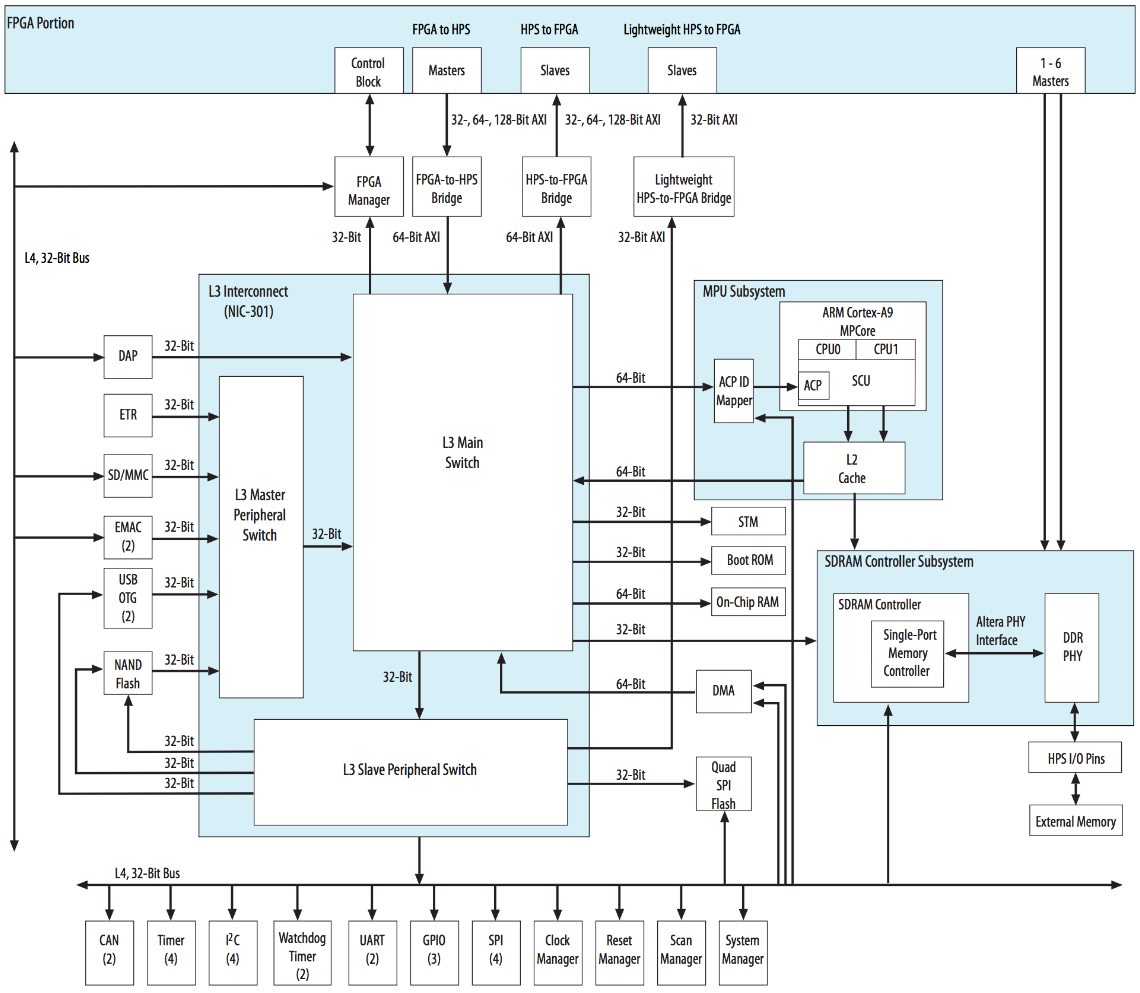


Figure 2.3: The HPS block diagram from the manual [5]

Some pins of the HPS, such as connections for the Light-Emitting Diodes (LEDs) on the Development Board or General Purpose Input Outputs (GPIOs), are physically tied to the HPS. By loaning these pins to the FPGA, it is possible to forward their functionality. By enabling this in Qsys, the concerned pin is routed through the HPS to the FPGA. A new pin assignment for the FPGA is not necessary.

The Field-Programmable Gate Array

The *Field-Programmable Gate Array (FPGA)* is the second element in the SoC and has a maximum global clock frequency of 460 MHz but Phase Locked Loops (PLLs) can be used to change clock frequency for single components within the FPGA's logic. The FPGA is programmed using a Raw Binary File (rbf), which is generated by Quartus. This configuration can be directly done via an USB blaster within Quartus, or via several SoC internal solutions, which can be chosen in the BSP-Editor.

The internal Interfaces based on AXI

The HPS and the FPGA communicate via interfaces, as seen in Figure 2.3, is based on the AXI standard. These interfaces can be distinguished between a Master on the one and a Slave on the other. Both sides are capable of transmitting and receiving, but the Master has to request an answer from the slave before the slave is able to write to the bridge in return. Always the first name in the name of the bridge is the master, whereas the second is the slave. The HPS-To-FPGA bridge (H2F) and the FPGA-To-HPS bridge (F2H) are capable of 32, 64 or 128 bit data width, whereas the Lightweight HPS-To-FPGA bridge (LH2F) can only transmit 32 bit at once. Each of these bridges can be enabled and disabled in Qsys, where the desired data width for the H2F and F2H can be adjusted as well. Inside the FPGA the *Avalon-MM* connects components inside the FPGA with the AXI bridges. Figure 2.4 visualizes this concept.

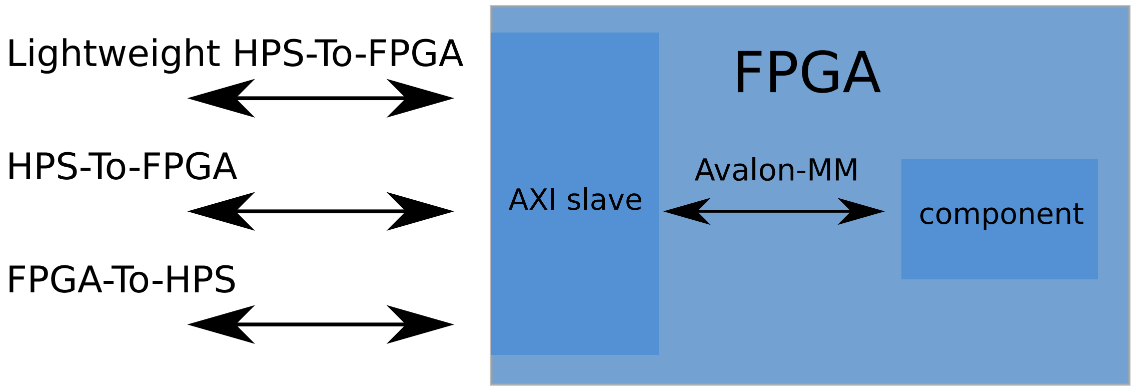


Figure 2.4: Blockdiagaram showing the concept of Avalon-MM

The *Avalon-MM* supports the implementation of read and write interfaces. The interface has several different signals, but the signals most important for this thesis are: *read*, *readdata*, *write*, *writedata* and *address* [2]. The signals *read* and *write* are both 1 bit signals, which is be set automatically by the interface each time the master of the interface wants to write to or read from the interface. Only if this enable bit is set, the data stream, transmitted by *readdata* and *writedata*, can be written to the interface. One AXI bridge can function for many The same AXI bridge can be used for multiple purposes in the same implementation. To distinguish each individual data set, a different *address* signal is used. When no address is set, the first address block inside the address space is used by default. The address space is discussed in Section 2.1.5.

2.1.2 Development Toolchain

This section describes all the tools used to program the FPGA. Figure 2.5 gives an overview about how different parts interact to generate the preloader.

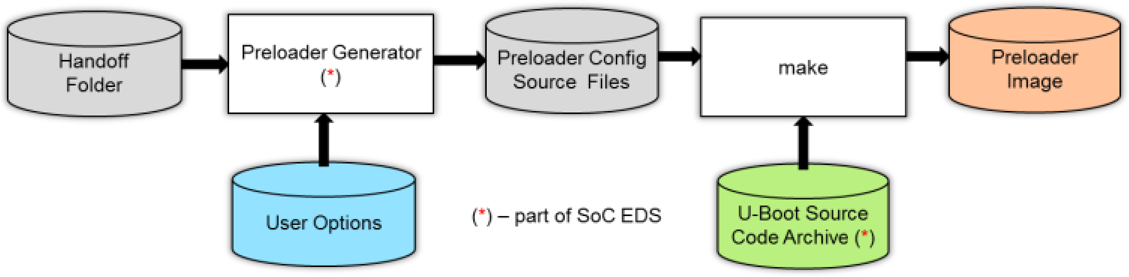


Figure 2.5: The SoC tool flow [19]

The integration tools Quartus and Qsys

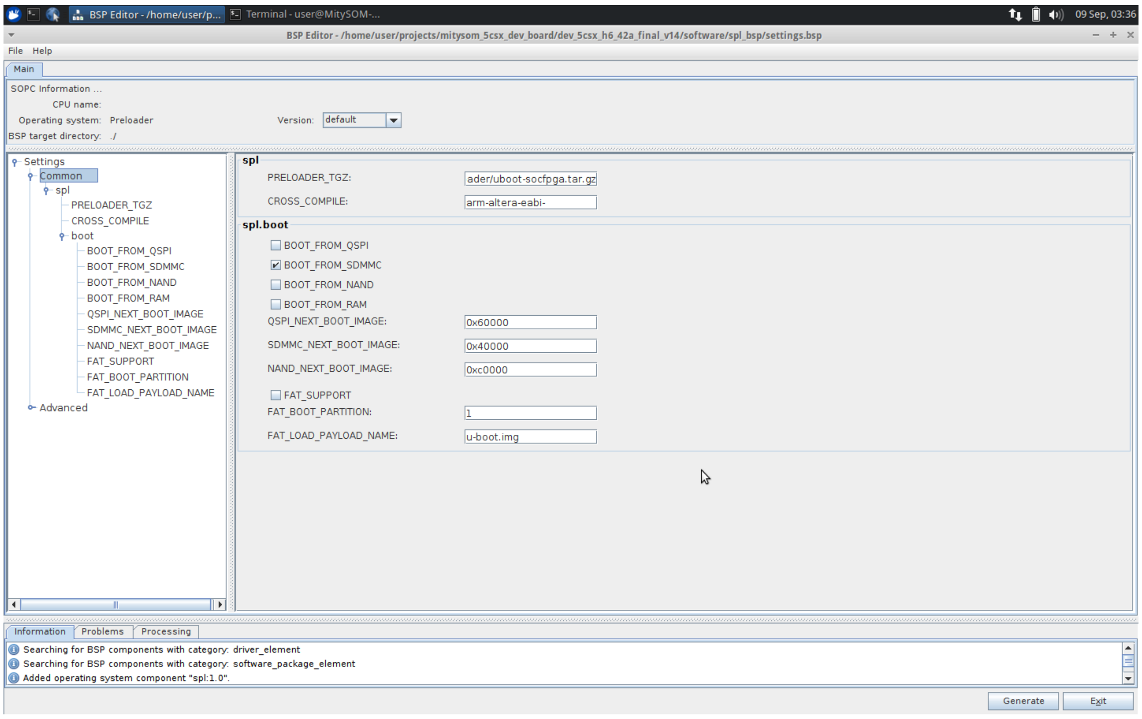
Quartus [**Quartus**] is the IDE provided by Intel for programming the FPGA's logic in Verilog or Very High Speed Hardware Descriptive Language (VHDL). The program was developed by Intel/Altera and the Premium Lite Edition comes with every Intel FPGA within the time of warranty for free. Each Quartus project has a *Top-Level Design* file, which describes a specific programmed behavior to be implemented by the FPGA's logic. This file can also integrate other HDL codes to use their implemented functionalities. After compiling the whole project, Quartus can be used to generate the rbf, a bitstream to program the FPGA. The compilation creates also a handoff directory. The subfiles in this folder are used by the BSP-Editor to generate the preloader. The *LOANIO* can be described as two 67 bit signals *loan_in*, *loan_out*, and one 1 bit signal, *loan_oe*. The first two are implemented as standard logic vectors with 67 entries, whereas the LOANIO output enabler is a 1 bit signal, that is only used to enable the communication. The enable bit has to be set manually, in order for the FPGA being able to write to use the loaned pin.

Qsys, as an integration tool in/for Quartus, helps the developer to build the HDL code. An comprehensible Graphical User Interface (GUI) makes it easy and intuitive to connect components. Each component represents a hardware subcircuit that is available as a library component for use in the Qsys tool [1]. Components can be added and removed from the project and their preferences adjusted. Qsys outputs a Quartus intellectual property (qip) file, which will be integrated into the Quartus project and the top-level design file. This qip contains subfiles to describe the logic of the used components and several tcl-Scripts, that are used by Quartus to solve for example the pin assignment for the user.

To add new functionality to a Qsys project, existing ones must be adjusted, or a new component added to the system. Qsys entails already pre-defined components by Altera, but it is also possible to ingerate an own component, based on HDL code. One of the most important pre-defined components is named *hps_0*, and it models the connection to the HPS. As mentioned in Section 2.1.1, it is possible to allow the FPGA to use peripheral from the HPS. In order to enable the FPGA access to GPIO50, LOANIO50 has to be enabled within Qsys. In the settings of the HPS component the required bridges can be enabled and adjusted as well.

Building Preloader and uBoot with the BSP-Editor and Generating the uBoot Environment

Quartus contains also the sopc-create-header-files tool and the BSP-editor, which can be accessed via the Altera Embedded Command Shell. Both tools use the hand-off files Quartus creates during the compilation. The Bootloader Support Package (BSP)-editor has been used to adjust the settings of the preloader, which is explained in Section 2.1.3. With the BSP-editor important options like SDRAM_SCRUBBING and Serial Support can be en- or disabled. After adjusting the BSP-editor generates a makefile and the user can create the preloader with *make* and uBoot, explained in Section 2.1.3, with *make uBoot*.



BSP Editor - /home/user/projects/mitysom_5csx_dev_board/dev_5csx_h6_42a_final_v14/software/spl_bsp/settings.bsp

```
SOPC Information ...
         CPU name:
 Operating system:  Preloader                Version: default
BSP target directory:  ./
```

```
spl
    PRELOADER_TGZ:              ader/uboot-socfpga.tar.gz
    CROSS_COMPILE:              arm-altera-eabi-

spl.boot
    [ ] BOOT_FROM_QSPI
    [✓] BOOT_FROM_SDMMC
    [ ] BOOT_FROM_NAND
    [ ] BOOT_FROM_RAM
    QSPI_NEXT_BOOT_IMAGE:       0x60000
    SDMMC_NEXT_BOOT_IMAGE:      0x40000
    NAND_NEXT_BOOT_IMAGE:       0xc0000
    [ ] FAT_SUPPORT
    FAT_BOOT_PARTITION:         1
    FAT_LOAD_PAYLOAD_NAME:      u-boot.img
```

```
Searching for BSP components with category: driver_element
Searching for BSP components with category: software_package_element
Added operating system component "spl:1.0".
```

Figure 2.6: Screenshot of the BSP-Editor [5]

To tell uBoot where to load the necessary files from, the Boot environment has to be generated out of the software/spl_bsp directory created by the BSP-Editor, with the *uBootMMCEnv.txt* and the command *uboot-socfpga/tools/mkenvimage -s 4096 -o ubootenv.bin uBootMMCEnv.txt*. The mentioned text file ships as a part of a VM, which is explained in the next section, Section 2.1.2.

The CriticalLink Virtual Machine and SD image

The manufacturer of the MitySOM-5CSX board Critical Link offers on their support page [16] an image of a whole Virtual Machine, which contains a xUbuntu 14.04 system with several programs pre-installed, such as Quartus, Qsys or Eclipse. CriticalLink provided a reference design implementation for the MitySOM-5CSX board in the form of a Quartus Design File (.qdf) and a Qsys project (.qsys). The pre-configured Quartus project has all the pins already assigned and it contains a pre-build top-level VHDL code. The problems accrued using this Virtual Machine (VM) are explained in Section 5.2.

Critical Link provides a 4 GB SD image specifically adapted to this board. This image is shipped with a Linux operating system, a complete file system and a few programs to test the proper functioning of the board saved in the /home/root directory.

It is also possible to build a custom SD card image. For that purpose Critical Link offers a script (*make_sd_card_shell.sh*) [**CLwikiSD**] on their support page. To build the image the shell needs to be in the same folder/directory with the preloader (preloader-mkpimage.bin), uBoot (u-boot.img), the uBoot environment binary (ubootenv.bin) and a tar ball of the root file system with compiled Device Tree Blob (dtb) and kernel in the /boot directory, called *rootfs.tar.gz* [16]. Preloader, uBoot and uBoot Environment can be created as shown in Section 2.1.2 The problems I had using this custom SD image are described in Section 5.2.

2.1.3 Boot Sequence

The preloader is a part of the BSP.

The booting procedure of the SoC follows a specific *boot flow*.

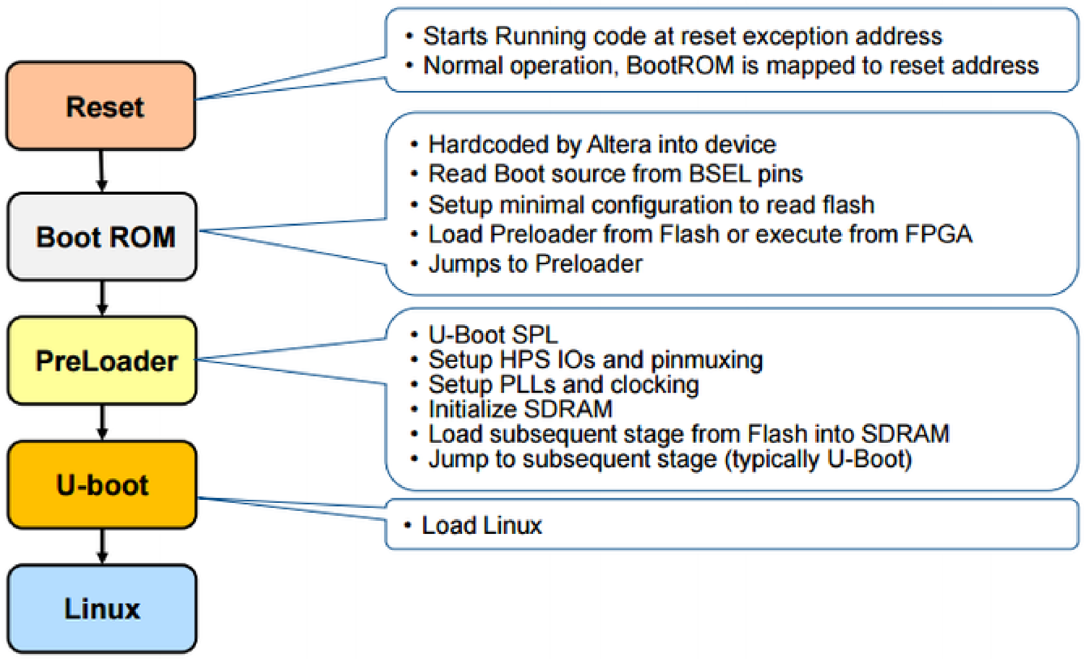

Reset
Boot ROM
PreLoader
U-boot
Linux
• Starts Running code at reset exception address
• Normal operation, BootROM is mapped to reset address
• Hardcoded by Altera into device
• Read Boot source from BSEL pins
• Setup minimal configuration to read flash
• Load Preloader from Flash or execute from FPGA
• Jumps to Preloader
• U-Boot SPL
• Setup HPS IOs and pinmuxing
• Setup PLLs and clocking
• Initialize SDRAM
• Load subsequent stage from Flash into SDRAM
• Jump to subsequent stage (typically U-Boot)
• Load Linux

Figure 2.7: The steps of the booting process [**RBEmbedded**]

After an external reset or a power-cycle, the board starts with the *Boot ROM*, which is hardcoded into the hardware by the manufacturer and prepares the board for the preloader, loading the preloader from the SD card into the RAM and jumping to the preloader. The preloader, as the last hardware setup step, sets all the pins and clocks of the FPGA. It is also responsible to program the FPGA's logic described in the rbf. After the preloader has prepared the hardware, uBoot cares about setting up the HPS for the Linux system by loading all the necessary files from the SD card.

The preloader does not need to be rebuild for every hardware modification. As shown in this section, the preloader depends only on the hardware initialized by the preloader through the rbf, or in other words on the hardware used by the FPGA and its configuration. Since most of the FPGA's functionality is described in Qsys, it mostly depends on the Qsys project, whether new hardware will be used, or it's preferences had been adjusted so that a new preloader has to be generated and exchanged with the old one on the SD card. Minor changes inside the logic, describing VHDL code of a component, will not lead to a necessary exchange of the preloader. A new component, which uses a new pin or interface, will entail to change it. How the preloader is generated is explained in Section 2.1.2.

2.1.4 Changing the Preloader and FPGA Reconfiguration

As described in Section 2.1.3, the preloader initializes the hardware, but the actual configuration of the FPGA is described in the rbf. To reconfigure the FPGA, the new rbf has to be copied to the SD card. If a new preloader is needed, is described in Section 2.1.2. To actually change the preloader, the developer has to load it manually on the third partition of the SD card, before the SD card is ejected from the computer. Therefore the command *dd if=preloader-mkpimage.bin of=/dev/sdX3* can be used.

To re-configure the FPGA one can either change the rbf in the /home/root directory on the SD card or use a cat/dd command within the Linux system. For the first way the user can either plug the SD card into a computer, or use the ARM to copy the new rbf into the /home/root directory on the SD card. If the computer is used, the preloader boots with the new configuration, but if the ARM was used to change the configuration, the board must be restarted. In both cases the new rbf has to replace the old one with the same name. For the second way multiple rbfs are stored on the SD card simultaneously. After the board is fully booted, including the Linux system, the FPGA can be re-programmed with *cat dev_5csx_h6_42a.rbf > /dev/fpga0* or *dd if= dev_5csx_h6_42a.rbf of=/dev/fpga0*. Exchanging the rbf manually on the SD card is the more laborious and slower way than using the Linux system on the ARM, if multiple rbf are to be tested. The second way requires however a Linux system running on the ARM, whereas the first way does not need the ARM at all.

2.1.5 Data Communication between FPGA-Module and AXI Bridge using mmap

Memory Mapping is a concept to access peripherals within a virtual address space, the same way the Central Processing Unit (CPU) can access memory blocks of its RAM. Therefore the whole memory and every additional external peripherals are combined in one bus and the RAM and each peripheral can be accessed through its given address space. This concept is visualized in Figure 2.8. Inside one address space different elements can be distinguished by different offsets. This value depends on the interfaces used by the FPGA and can be generated with a tool after Quartus compiled its whole project. This *sopc-create-header-files* tool, provided by Altera, is included into Critical Links VM and can be accessed through the Embedded Command Shell.

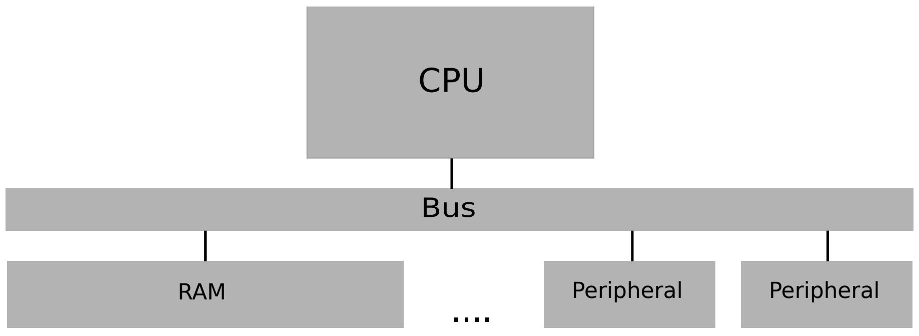


Figure 2.8: The concept of memory mapping [15]

So in order to actually communicate through this bride, each side has to address the bridge correctly. On behalf of the FPGA, an existing component has to implement the Avalon interface correctly. An example of how this could be implemented is shown in Section 4.1.1. The HPS uses memory mapping to write to the right address space. Therefore a C code / program running on the Linux system using *mmap()* only needs the base address and offset.

2.2 The Eclipse plugin

As stated in Section ..., an IDE plugin should be developed to support the preparation of hardware accelerators from java programs. In Section ..., the Eclipse IDE [...] is chosen for this purpose. This section outlines the generic Eclipse plugin development process.

2.2.1 Basic Concepts and the Eclipse IDE for Eclipse Committers

Eclipse is based on the Equinox framework, which is responsible for the plugin management and execution [8]. Since a plugin can not initiate anything by itself it needs always the Equinox framework to start the Plugin or its functionality [8]. Every eclipse functionality, such as code editors with syntax highlighting and auto-completion, is realized by a plugin. Plugins can interact with the user through very different elements such as context menus or tabs inside the Eclipse GUI and are offering it's functionalities to the framework - or to other plugins - through it's Extensions, as explained in detail in the next section.

To support the development of an Eclipse plugin, the Eclipse Foundation has created a package called „Eclipse IDE for Eclipse Committers" [6]. This package basically is focused on Java development, but it includes also several functionalities such as presets and tutorials to develop an Eclipse Plugin, a Git integration or the Eclipse Extensible Markup Language (XML) editor. The Eclipse PlugIn Development Environment (PDE) is a Plugin which basically provides similar functionality but the last version is from 2013 and it is missing the tutorials.

2.2.2 The Manifest File and how Equinox is using it

The main file of every Eclipse Plugin is the Manifest File *manifest.mf.* This file is used to specify, how this plugin is handled by the Equinox framework. During the installation of the plugin this file will be parsed and the plugins structure will be embedded into the framework. This helps the Eclipse instance to decide when to launch and kill a plugin. This is related to the plugins lifecycle [7], a topic that is out of the scope of this thesis.

The manifest.mf file (see Listing 2.1) has nine subsections: Overview, Dependencies, Runtime, Extensions, Extension Points, BUILD, MANIFEST.MF, plugin.xml, and build.properties.

Extensions and *Extension Points* are used to extend existing plugins. The Extension Point provides conditions to offer its functionality and only a plugin with a fitting Extension can use it [3]. After adjusting the MANIFEST.MF the Extension and Extension Point appear as XML elements in the plugin.xml.

Listing 2.1: Utilized manifest.mf file

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: JavaFPGAEditor
```

```
Bundle-SymbolicName: de.ceglarek.plugin.javaFPGAEditor;singleton
    :=true
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Require-Bundle: org.eclipse.jdt.core;bundle-version="3.13.0",
 org.eclipse.core.runtime;bundle-version="3.13.0",
 org.eclipse.core.resources;bundle-version="3.12.0",
 org.eclipse.core.expressions;bundle-version="3.6.0",
 org.eclipse.e4.core.di;bundle-version="1.6.100",
 org.eclipse.e4.ui.services;bundle-version="1.3.0",
 org.eclipse.ui
Import-Package: org.eclipse.jface.dialogs,
 org.eclipse.jface.viewers,
 org.eclipse.swt.widgets
```

The *plugin.xml* (see 4.1) is actually the code version of what has been defined in the tabs ”Extensions” and ”Extension Points”. This file is used for wrapping the whole Plugin into a Java Archive (JAR) Packaging.

3 Related Work

This section will give an overview about already existing conversion tools, what their key features are. Further I will do the same for HLS tools.

A final comparison and evaluation of these tools has been done in Chapter 5.

3.1 Running Java within a JVM usingJOP

Java byte-code can be run directly on an FPGA using a Java Virtual Machine (JVM) implemented in hardware. This method is not very performant, because the hardware has to emulate the VM, as well. Using a JVM does however not require to use a HLS tool.

The Java Optimized Processor (JOP) [20] is an implementation of the JVM in hardware. The main implementation platform is an FPGA. JOP is a time-predictable processor for hard real-time systems implemented in Java. It was published in 2009 and supports Java version 1.4. JOP is open-source under the GNU's Not Unix! (GNU) GNU General Public License (GPL) version 3 and has a growing user base.

3.2 Existing Java to C converter

There are already a few solutions for converting Java to C code. This C code is meant to be converted again by existing HLS tools, which are represented in Section 3.3.

3.2.1 JCGO

JCGO [14] is a software application which translates programs written in Java into platform-independent C code. JCGO translator uses some optimization algorithms that allow, together with optimizations performed by a C compiler, the resulting executable code to reach better performance if compared with the traditional Java implementations (based on the Just-In-Time technology). JCGO was last updated 2014 and supports Java v1.4.

3.2.2 JC

JC [4] offers a conversion of Java byte-code into C. It includes a Java byte-code interpreter with support for compiled, interpreted, and mixed mode execution

Its was last updated 2005 und supports Java version 1.2 and is covered by GNU GPL version 2.

3.2.3 Toba

Toba [27] is a system for generating efficient standalone Java applications. Toba includes a Java-bytecode-to-C compiler, a garbage collector, a threads package, and Java API support. Toba-compiled Java applications execute according to its documentation [26] 1.5–10 times faster than interpreted and Just-In-Time compiled applications. Its current version 1.1c was released 1999. It development stopped with Java Development Kit (JDK) 1.1. Although it is also a stand alone Java to C compiler, it is primarily built to increase the performance of Java written code by pre-compiling Java class files into C code and this C code directly into machine code.

3.2.4 Varycode

Varycode [23] is an online code converter which provides several convert options. Among others the user can convert Java to C#, Visual Basic .Net, Ruby, Iron Python or Boo. It supports enums, classes, interfaces, generic methods and classes, collections, exception handling, annotations, native API functions declaration, try constructions, loops, switches, String, char and many JRE library classes. Varycode is royalty free for up to 2048 input chars. It is not obvious which Java version the tool supports.

3.2.5 Conversion from Java to C++

Although it is not directly linked to the scope of this thesis, because it converts Java to C++, the J2C project [21] should also be mentioned. J2C is an open source code converter, that expresses its functionality as an Eclipse plugin. It works with every Java version 1.6 and was last updated 07.07.2015.

3.3 High-Level Synthesis (HLS) tools

HLS tools are used to translate functionalities from one programming language into another. The aim is to interpret the behavior of a code and implement this functionality in hardware. Because of its natural close relation to hardware C is the most commonly used input language for a HLS tool.

HLS tools are used to

3.3.1 LegUp

The University of Toronto developed the open-source HLS tool LegUp [13]. It is specialized to synthesis C to Verilog and was developed with the aim to make FPGA programming easier. LegUps last update was on 24 August 2015 with version 4.0 [11]. It is for non-profitable usage free but they also launched a company called LegUp Technologies Inc. [12].

3.3.2 SPARK

The [22] HLS tool was developed by the Microelectronic Embedded Systems Laboratory University of California San Diego. It uses ANSI C as an input and converts it to VHDL.

3.3.3 Vivado HLS

Vivado [24] is a HLS tool developed by the FPGA manufacturer Xilinx. It ships with their product for every in-warranty user. Vivado can handle both C and C++ files as an input and outputs either VHDL or Verilog.

Figure 3.1 [25] shows the concept of the conversion.

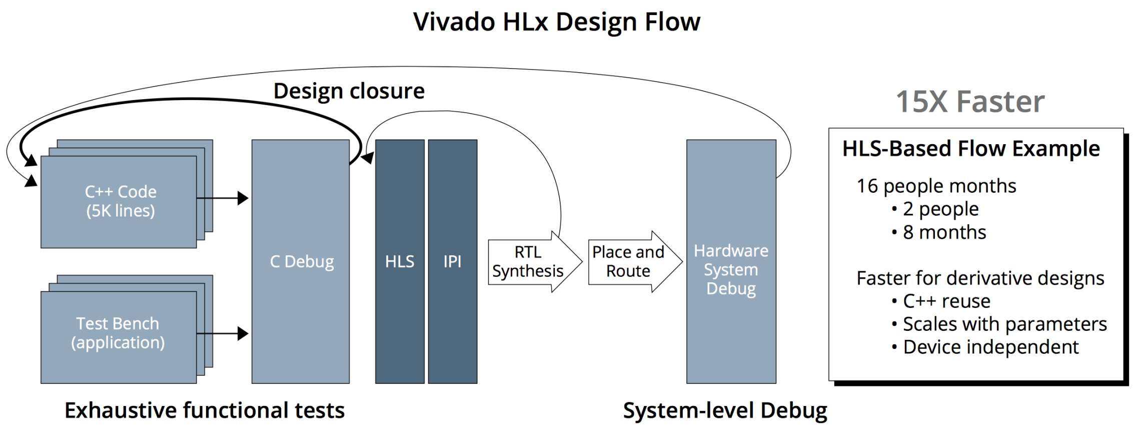


Figure 3.1: Block Diagram of the Vivaldo Design Flow

3.3.4 Intel HLS Compiler

The Intel HLS Compiler [10], developed by Intel, is only available as beta access by the time of this thesis. It handles untimed ANSI C++ files and is meant to work directly as an input for the Qsys integration tool.

3.4 Analysis of existing solutions

Although there are different approaches to run Java code on an FPGA, none of them fits the needs of the OPS-Sat project. Running Java code natively on the FPGA using a JVM results not only in poor performance, since the FPGA has also to implement the VM in hardware, but additionally misses the requirements of this thesis. The FPGA should act as a hardware accelerator and the SoC must also be able to execute C code. Furthermore non of these tools works with Java code based on versions more recent than 1.5. So there is no solution for recent code. However parts of existing tools can be used by this plugin in further projects. The HLS tools are useful, when Java code has already be converted to C. Since however none of them can handle Java files, they do not directly fulfill the requirements, but can be used to generate HDL code, when the Java code has been converted to C.

4 Implementation

Because there is no fitting solution to hardware-accelerate Java code on the OPS-Sat's MitySOM-5CSX FPGA, this thesis is about to implement the necessary features in hardware and to start developing the inherent IDE plugin, that helps the developer to prepare their Java code to be hardware-accelerated by the FPGA.

4.1 Example for Hardware-Kernel Controlled by HPS

As described in Section 2.1.2, Critical Link provides its customers with an image of a highly equipped VM, which also includes an example implementation for the MitySOM-5CSX. Based on this Quartus and Qsys project, a simple example for an FPGA hardware accelerator has been implemented in two steps. The first example uses only the FPGA to demonstrate the basic functionality of the FPGA and the used development chain. The second example uses the HPS and additionally one AXI bridge, to setup a fully functioning interface for the communication between HPS and FPGA. To make the result visual, these examples control an LED on the MitySOM-5CSX development board.

4.1.1 Modifications for LED-Blinky

Example One: Standalone LED blinker driver

The Qsys modules provided by the Critical Link example project, described in Section 2.1.2, are pre-defined, so that their logic can be changed. A custom component has been written in VHDL to implement the requested logic. This new LED component uses the 100 MHz signal from a clock component in Qsys and outputs a 1 Hz signal with the help of a 27 bit counter. So the most significant bit toggles at $2^{27}/100\,\text{MHz} = 1.34\,\text{s}$ It has a reset and clock signal as input signals and LED as an output signal. Finally, in the port mapping section of the code, the highest bit of this vector gets linked to the one bit LED output signal. In Qsys the component has to be connected to a clock and a reset signal and the LED signal has to be exported, so that the signal appears as an output signal after the project has been compiled.

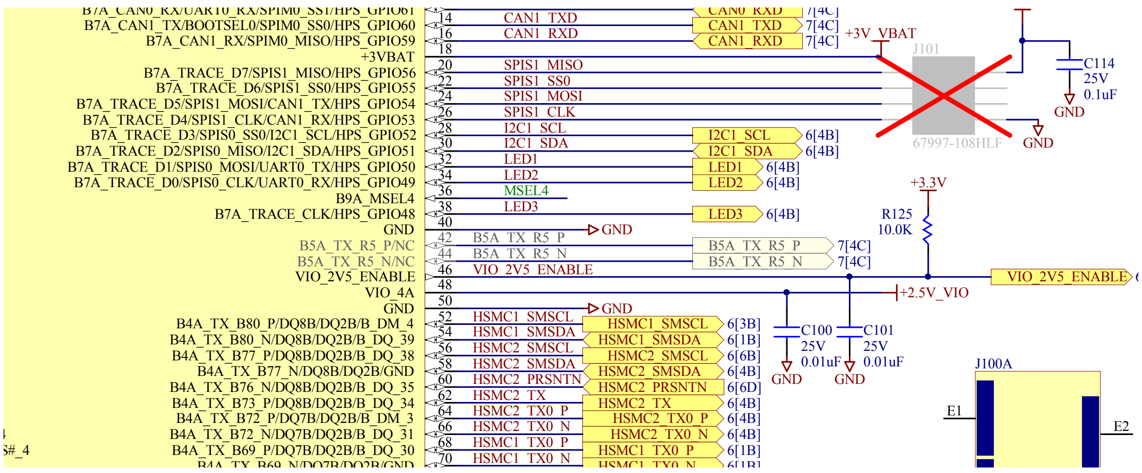


Figure 4.1: The schematics of the used development board

The schematics of the development board, as shown in Figure 4.1, shows, that GPIO50 is the pin for *LED1*. The LED is physically wired to the HPS. So in order to enable the FPGA access to this pin, the corresponding GPIO50 must been loaned to the FPGA, as described in Section 2.1.2.

In order to communicate with the HPS or its pins, the FPGA can use one of AXI bridges described in Section 2.1.1. Each of the three bridges has its own use case and according to the manual [5] the lightweight interface is most useful for accessing the control and status registers of soft peripherals and useful to low-bandwidth traffic, because it has a fixed bandwidth and a small address space [**c5manLWB**]. So the Lightweight HPS-To-FPGA bridge is best suited for this implementation, since the transmitted information is very small and the communication will be mastered by the HPS.

Modification of the Top-Level Design File To integrate the new components correctly into the project, several changes in the top-level design file are necessary. The following paragraph explains how the different VHDL codes interact with each other, visualized by Figure 4.2, and what changes has been made to the provided Quartus project.

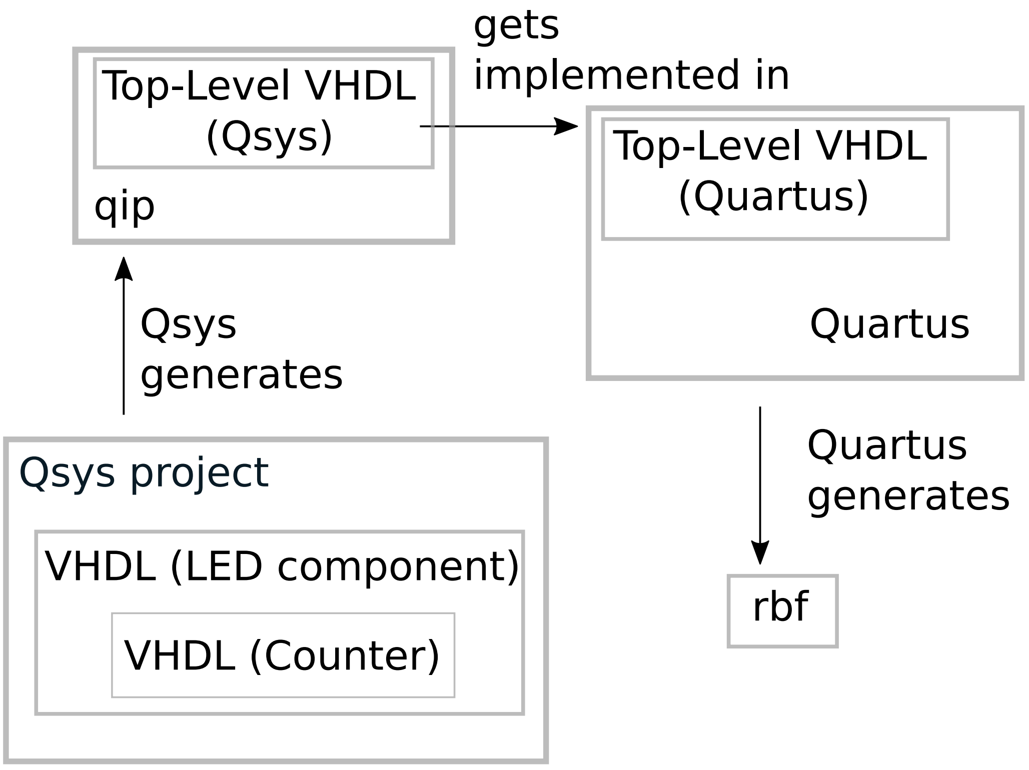


Figure 4.2: How the different VHDL codes interact with each other

After Qsys compiled the project into a qip file, this file has to be integrated into the Quartus project. Within the qip is a VHDL, describing the implemented logic. This code needs to be implemented into the predefined top-level design VHDL code. The Qsys VHDL file has the led signal as an output, which has to be added to the Quartus VHDL code. Additionally all the GPIO entries, which were adjusted as LOANIOs in Qsys, have to be renamed to *"LOANIO"* manually. Because it is not possible to modify imported signals, a signal has to be created and mapped to the exported LED signal. Conclusively the LED signal has been routed through the HPS to physical LED, using LOANIO50. Finally the loan enable signal has to set to ‚1‘, as described in Section 2.1.2.

After compiling the Quartus project into a rbf, the BSP-Editor is used to generate the preloader. The FPGA has to boot from the SD card, so this has to be enabled.

Finally the rbf has to be copied on an existing SD card or a new SD card image has to be created, like explained in Section 2.1.2.

Example Two: HPS and FPGA using LED blinker driver

The second examples adds the HPS and presupposes, that the Linux system provided by Critical Link runs on the ARM, which is explained in Section 2.1.2. The aim is to control the same LED used in the first implementation with inputs to a program running on the HPS given by the user. The C program on the ARM uses *mmap* to address the communication bridge to the FPGA, as shown in Figure 2.1.

Modifications made in Qsys Based on the project of the first implementation, the logic to the LED component needs to be extended. The extended component includes an Avalon-MM interface and the logic to react to data transferred via this interface.

The *led_blinker* VHDL code obtains the necessary signals *write*, *writedata*, *read* and *readdata* to implement the Avalon-MM interface, as described in Section 2.1.1. The bitstream in *writedata* gets linked to a *step* register, which changes the counting speed. The *write* signal is set, when the master writes to AXI bridge/interface, as shown in Section 2.1.1. The FPGA component perceives when the signal is set and then reads out the value in *writedata*. This value changes the speed of the *counter* from the first example.

To connect the added interfaces in the VHDL code to the Qsys project, the Avalon-MM interface has to the added and adjusted in the editing tool of the Qsys component. Qsys detects the right interface automatically based on the names of the signals.

The C program running in the ARM using mmap On the other of the bridge, the ARM uses *mmap* to address the AXI interface. The principle is, that the user starts the C program, which transmits a certain value through the selected AXI bridges to the FPGA, where it changes the counter frequency.

For this thesis a cross compiled C program running on the HPS sends 0x00000000 or 0xFFFFFFFF to the FPGA and the FPGA toggled the LED on the development board. 0x00000000 causes the LED to hold its current status, either being on or off, and 0xFFFFFFFF sets the LED to a 2 Hz blinking frequency. *mmap* needs the base address and the offset, to address the selected bridge. They were generated out of the compiled Quartus project, as described in Section 2.1.2.

For the lightweight bridge a base address of 0xFF200000 with a span of 0x00200000 can be found in the manual of the FPGA. The offset inside the address space can be generated out of the Quartus compiled project as described in Section 2.1.5. The generated header file contains the name, the start address inside the relevant address space, the span and end address of every component connected to the AXI bridges. *mmap* uses pointer and since the lightweight bridge has a 32 bit data width, a 32 bit pointer was used. To run the C program on the ARM processor it was cross compiled with the GNU cross compiler.

In addition to the existing functionalities a bit inverter has been developed, to demonstrate a usecase of the address signal and that the FPGA can truly change transmitted data and send it back to the master. The FPGA bitwise inverts the transmitted values and uses a second *address* of the lightweight bridge and so it writes to the next 32 bit of address space as described in Section 2.1.1.

4.2 The Eclipse Plugin

Although the development of the plugin could not be finished, because of problems mentioned in Section 5.2, a simpe example plugin had been realized. The *Eclipse IDE for Eclipse Committers* package [6] has been used. How an Eclipse plugin works and what the plugin.xml file is, is explained in Section 2.2.

This example plugin uses two official Eclipse extension points to access it via the GUI and open an editor (see 4.1).

Listing 4.1: Utilized plugin.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
        <extension
                point   ="org.eclipse.ui.menus">
                <menuContribution locationURI="popup:org.eclipse
                    .jdt.ui.PackageExlorer">
                        <command
         commandId="de.ceglarek.plugin.htmlconverter.convert"
         label="Create␣@FPGA␣Annotation"
         style="push">
                        </command>
                </menuContribution>
        </extension>


         <extension
                point="org.eclipse.ui.commands">
        <command defaultHandler="de.ceglarek.plugin.
```

```
            javaFPGAEditor.editors.EditorClass"
             id="de.ceglarek.plugin.javaFPGAEditor.FPGAAnnotation
                " name="Convert">
        </command>
    </extension>

     <extension point ="org.eclipse.ui.editors">
                <editor
                        name="Properties␣Editor"
                        extensions="mpe"
                        contributorClass= "de.ceglarek.plugin.
                           javaFPGAEditor.editors.EditorClass"
                        class="de.ceglarek.plugin.javaFPGAEditor
                           .editors.EditorClass"
                        id="de.ceglarek.plugin.javaFPGAEditor.
                           EditorClass">
                </editor>
        </extension>
</plugin>
```

The first extension point (*org.eclipse.ui.commands*) creates a new menu entry in the context menu in the workspace. The handler referes to the Java class, where the actual functionality of the plugin is described. The second extension point (*org.eclipse.ui.editors*) opens an editor to manipulate the code.

5 Results

5.1 The Configuration of the FPGA, Analysis of the Existing Solutions and the Plugin Examples

The two example implementations show in detail how a complete hardware-accelerator would be implemented into the hardware. The FPGA can be re-configured only using the ARM. The reconfiguration however does require to power-cycle the board.

The analysis of existing solutions to execute Java code on the FPGA fabric showed, that none of them fulfilled the requirements defined by the project.

For the development of new Eclipse plugins, a complete package (*Eclipse IDE for Eclipse Committers*) is provided by the Eclipse Foundation. This package makes it easy to develop new plugins and it includes several examples and tutorials. The created example demonstrates how a new context menu entry can be integrated to the GUI to launch the plugin.

5.2 Problems

While working on this thesis technical problems accrued, most of them caused by the provided VM.

5.2.1 Problems with the provided VM

The VM provided by CriticalLink runs, although it was tested on various computers, unstable. Often the whole OS stopped operating or the GUI were so slow, that it was impossible to work with. Sometimes only a complete restart of the VM, without saving the current status, solved the problem.

Besides the Qsys conversion time with approximated 20 minutes and the Quartus compilation time with over 35 minutes the permanent hang-ups of the Quartus software caused tremendous time problems. Often the software did not responed for a few minutes or other bugs were causing problems. And there were other problems with Quartus, as well. In order to compile the Quartus project, the right HDL file had to be set as the top-level design file. Quartus switched sometimes this setting without any influence on behalf of the user. Also adding the compiled qip file often crashed Quartus. The converted qip file should be added automatically to the Quartus project, but every ones in a while it switches to manually without any obvious reason.

ESA required to use this VM, so that every developer uses the same tool set. This simplifies the support.

5.2.2 Problems with the SD image

In order to fully understand the operating system running on the ARM, I wanted to build a custom Linux SD card image based on the instructions from the Critical Link support page [16] and the rocket board tutorials [**RBEmbedded**].

The custom SD card image however wasn't that easy to generate even with a lot of help from the university's tutor. Some problems remain unsolved, like a problem with re-programming the FPGA with the new rbf using the Linux on the SD card only. While with the provided system one can use the cat command, using this on the custom one results in a crash of the board, which requires a hardware reset. An ethernet driver error also displayed during booting the board.

6 Conclusion and Outlook

Despite all the problems described in Section 5.2, a communication between the FPGA and the ARM could be implemented and both examples achieved the required results. The second, more sophisticated example implemented a communication interface like it will be used in the fully automated hardware-accelerator. The implementation of the Eclipse plugin however could not be finished because of reasons described in Section 5.2. Nevertheless a simple example plugin had been developed, which uses official Eclipse extension points to add a menu entry. Future projects can build on this implementation.

The concept to use an IDE plugin to convert selected segments of Java code to VHDL leads to a very flexible solution. The plugin helps the developers to prepare their code for a conversion directly in the IDE, which is very convenient. Future work will build on the FPGA implementation and the research made in this thesis regarding the existing solutions. The example plugin shows how an Eclipse plugin basicially works.

List of Figures

List of Tables

List of Acronyms

ARM	Advance RISC Machine
ANSI	American National Standards Institute
AXI	Advanced eXtensible Interface
BSP	Bootloader Support Package
CPU	Central Processing Unit
dtb	Device Tree Blob
ESA	European Space Agency
FPGA	Field-Programmable Gate Array
GNU	GNU's Not Unix!
GPIO	General Purpose Input Output
GPL	GNU General Public License
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Systhesis
HPS	Hard Processor System
IDE	Integrated Development Environment
LED	Light-Emitting Diode
JAR	Java Archive
JDK	Java Development Kit

JOP	Java Optimized Processor
JVM	Java Virtual Machine
PDE	PlugIn Development Environment
PLL	Phase Locked Loop
rbf	Raw Binary File
qip	Quartus intellectual property
SoC	System on Chip
SOM	System on Module
VHDL	Very High Speed Hardware Descriptive Language
VM	Virtual Machine
XML	Extensible Markup Language

Bibliography

[1] Altera. *Making Qsys Components*. English. Altera Corporation - University Program. Aug. 2012. URL: http://scale.engin.brown.edu/classes/EN2911XF14/QSYS_COMP.pdf (visited on 05/10/2017).

[2] *Avalon Interface Specifications*. English. Intel Corporation. May 2017. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf (visited on 05/10/2017).

[3] N. V. Chris Laffra. *FAQ What are extensions and extension points?* English. The Eclipse Foundation. June 2006. URL: https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F (visited on 05/10/2017).

[4] A. L. Cobbs. *JC Virtual Machine - Welcome to JVC*. English. 2004. URL: http://jcvm.sourceforge.net (visited on 05/10/2017).

[5] *Cyclone V Hard Processor System Technical Reference Manual*. English. Altera Corporation. Oct. 2016. URL: https://www.altera.com/documentation/sfo1410143707420.html (visited on 09/07/2017).

[6] Eclipse. *Eclipse IDE for Eclipse Committers*. English. The Eclipse Foundation. URL: http://www.eclipse.org/downloads/packages/eclipse-ide-eclipse-committers/oxygenr (visited on 05/10/2017).

[7] Eclipse. *Plugin Lifecycle*. English. The Eclipse Foundation. URL: https://www.eclipse.org/che/docs/assemblies/plugin-lifecycle/ (visited on 05/10/2017).

[8] A. T. Gabriel Wetzler. *Eclipse Plugins*. Deutsch. Fern-Universität Hagen. 2010. URL: https://wiki.fernuni-hagen.de/eclipse/index.php/Plugins (visited on 05/10/2017).

[9] *Intel Cyclone V*. English. Intel Corporation. URL: `https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html` (visited on 09/07/2017).

[10] *Intel HLS Compiler*. English. Intel Corporation. 2017. URL: `https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html` (visited on 09/07/2017).

[11] *LegUp 4.0 Documentation*. English. University of Toronto. 2015. URL: `http://legup.eecg.utoronto.ca/docs/4.0/index.html` (visited on 05/10/2017).

[12] *LegUp Computing*. LegUp Computing Inc. URL: `http://www.legupcomputing.com` (visited on 05/10/2017).

[13] *LegUp High-Level Synthesis*. English. University of Toronto. URL: `http://legup.eecg.utoronto.ca` (visited on 05/10/2017).

[14] I. Maidanski. *JCGO: Java and C/C++ web developer resources*. English. IvMaiSoftLLC. Dec. 2015. URL: `http://www.ivmaisoft.com/jcgo/links.htm` (visited on 05/10/2017).

[15] H. Mao. *Exploring the Arrow SoCKit Part III - Controlling FPGA from Software*. English. URL: `http://zhehaomao.com/blog/fpga/2013/12/27/sockit-3.html`.

[16] *MitySOM-5CSx Altera Cyclone V SOC Wiki Page*. English. Critical Link LLC. URL: `https://support.criticallink.com/redmine/projects/mityarm-5cs/wiki/` (visited on 09/07/2017).

[17] *MitySOM-5CSxSingle or Dual Cortex A9 and User Programmable FPGA SOM*. Critical Link LLC. URL: `http://www.criticallink.com/product/mitysom-5csx/` (visited on 09/07/2017).

[18] *OPS-SAT*. Englisch. Apr. 2017. URL: `http://www.esa.int/Our_Activities/Operations/OPS-SAT` (visited on 09/07/2017).

[19] *Preloader and U-Boot Customization - v13.1*. English. v13.1. RocketBoards.org. Feb. 2017. URL: `https://rocketboards.org/foswiki/view/Documentation/PreloaderUbootCustomization131` (visited on 05/10/2017).

[20] M. Schoeberl. *JOP - Java Optimized Processor*. English. 2007. URL: `http://www.jopdesign.com` (visited on 05/10/2017).

[21] J. Sieka. *J2C Compiler*. English. July 2015. URL: `https://bitbucket.org/arnetheduck/j2c` (visited on 05/10/2017).

[22] *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Microelectronic Embedded Systems Laboratory University of California San Diego. URL: `http://mesl.ucsd.edu/spark/` (visited on 09/07/2017).

[23] *Varycode Startpage*. Varicode Inc. URL: `https://www.varycode.com/` (visited on 05/10/2017).

[24] *Vivado*. English. XILINX INC. URL: `https://www.xilinx.com/products/design-tools/vivado.html` (visited on 09/07/2017).

[25] *Vivado Design Suite HLx Editions*. English. Xilinx Inc. 2015. URL: `https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf` (visited on 05/10/2017).

[26] T. A.P.G.T.P.B.J.H.H.T.N.S. A. Watterson. *Toba: Java For Applications A Way Ahead of Time (WAT) Compiler*. The University of Arizona. URL: `ftp://ftp.cs.arizona.edu/sumatra/report/toba.pdf` (visited on 05/10/2017).

[27] T. P.J.H.G.T.P.B.P.B.T.N. S. Watterson. *Toba: A Java-to-C Translator*. The Sumatra Project. Apr. 1999. URL: `https://www2.cs.arizona.edu/projects/sumatra/toba/` (visited on 05/10/2017).