VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TESTING OF GENERATED C COMPILERS FOR PROCESSORS IN EMBEDDED SYSTEMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. LUDĚK DOLÍHAL
AUTHOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TESTOVÁNÍ GENEROVANÝCH PŘEKLADAČŮ JAZYKA C PRO PROCESORY VE VESTAVĚNÝCH SYSTÉMECH

TESTING OF GENERATED C COMPILERS FOR PROCESSORS IN EMBEDDED SYSTEMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE AUTHOR	Ing. LUDĚK DOLÍHAL
VEDOUCÍ PRÁCE SUPERVISOR	prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2016

Abstrakt

Vestavěné systémy se staly nepostradatelnými pro náš každodenní život. Jsou to obvykle úzce zaměřená, vysoce optimalizovaná, jednoúčelová zařízení. Jádro vestavěných zařízení obvykle tvoří jeden nebo více aplikačně specifických instrukčních procesorů. Tato disertační práce se zaměřuje na problematiku testování nástrojú pro návrh aplikačně specifických procesorů a následně i samotných aplikačne specifických procesorů. Snahou bylo vytvořit systém, ve kterém bude možné otestovat jednotlivé nástroje, jako například překladač, assembler, disassembler, debugger. Nicméně vyvstává také potřeba provádět složitější testy, například integrační, které zaručí, že mezi jednotlivými nástroji nevzniká nekompatibilita. Autor vytvořil s podporou průběžně integračního serveru prostředí, které napomáhá odhalování a odstraňování chyb při návrhu aplikačně specifických procesorů a které je navíc do značné míry automatizované.

Abstract

Embedded systems have become essential for our everyday lives. They are usually highly specialized and optimized single purpose devices. The cores of these devices are usually composed of one or more application specific instruction-set processors. This dissertation thesis focuses on testing of tools for design of application specific instruction set processors (ASIP) and ASIPs themselves. The aim is to create a system which allows testing of tools, such as a compiler, an assembler, a disassembler or a debugger. Nevertheless, there is also need for more complex tests, for example, integration tests which ensure there is no incompatibility between the tools. The author created, with the support of a continuous integration server, an environment that helps to reveal and fix errors during the design of application specific processors and, moreover, this environment is automatized up to a certain point.

Klíčová slova

Testování, překladače, průběžná integrace, hardware software codesign, procesory s aplikačně specifickou instrukční sadou, jazyky pro popis architektury, vestavěné systémy.

Keywords

Testing, compilers, continuous integration, hardware software codesign, application specific instruction set processors, architecture description languages, embedded systems.

Citace

Luděk Dolíhal: Testing of generated C compilers for processors in embedded systems, disertační práce, Brno, FIT VUT v Brně, 2016

Testing of generated C compilers for processors in embedded systems

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.

....
Luděk Dolíhal
November 19, 2016

Poděkování

Na tomto místě bych rád poděkoval svému školiteli profesoru Tomáši Hruškovi za jeho vedení, čas, rady a velkou podporu, kterou mi poskytoval během mého studia. Dále bych rád poděkoval kolegům, především Karlu Masaříkovi, Zdeňku Přikrylovi, Adamu Husárovi, Ondřeji Ilčíkovi, Liboru Vašíčkovi, Róbertu Baručákovi, Filipovi Matiovskému, Milanu Skálovi a dalším členům Codasip týmu za jejich skvělou spolupráci a nápady.
V neposlední řadě také velmi děkuji svým rodičům Františkovi a Janě Dolíhalovým a své přítelkyni Juliáně Krejčové bez nichž by tato práce nikdy nemohla vzniknout.

Contents

1 Introduction **3**

2 State of art **5**
2.1 Standard library . . . 5
2.2 Test-suites for the C compiler . . . 6
2.2.1 GCC test-suite . . . 7
2.2.2 LLVM test-suite . . . 7
2.2.3 Test selection mechanism . . . 8
2.3 Continuous integration . . . 8
2.3.1 Jenkins as a build environment . . . 9
2.3.2 Current possibilities of the job generation . . . 10

3 Lissom project **11**
3.1 CodAL Language . . . 11
3.2 Toolchain . . . 12

4 Porting of the C library **13**
4.1 Theory of Porting . . . 13
4.1.1 Special instruction principle . . . 15
4.1.2 Simulators . . . 15
4.2 Automation of the porting process . . . 16
4.3 Experimental results and contribution . . . 17

5 Tests selection **19**
5.1 Test selection scheme . . . 19
5.1.1 Test selection phase . . . 19
5.1.2 Test compilation and execution . . . 20
5.1.3 Logging information and test evaluation . . . 21
5.2 Generator of the test selection files . . . 21
5.2.1 Design of the generator of test selection files . . . 21
5.3 Experimental results and contribution . . . 23

6 Acceleration of testing **25**
6.1 Testing attitudes . . . 25
6.1.1 Testing oriented on tools . . . 25
6.1.2 Testing oriented on models . . . 26
6.2 Case study and experimental results . . . 27
6.3 Main contribution . . . 28

7 Continuous integration job generator **29**
7.1 Job generation 29
7.1.1 Sniffer 29
7.1.2 Templates 30
7.1.3 Job generator 30
7.2 Experimental results and contribution 32

8 Conclusion **33**
8.1 Future work 34

Chapter 1

Introduction

This thesis is going to deal with the area of hardware software codesign and will mainly focus on testing and stability of such tools. Every piece of software contains errors and tools for hardware software codesign are not an exception. It is a well-known fact that the later the error is discovered in the software, the more expensive the process of fixing it is.

In order to uncover bugs in the early stages of development, tools have to be tested. Usually the better the coverage of the environment, the more bugs are triggered and can be fixed. To uncover the bugs, quality assurance teams and teams that focus on development of internal tools put a lot of effort into the design of new testing approaches. Nowadays, the majority of testing is performed automatically by advanced continuous integration systems (CI systems). However, there are still testing scenarios that cannot be automatically tested. The human element cannot be omitted in the process of testing.

A testing system, especially one for a complicated integrated development environment, such as a tool for hardware software codesign [10], must be capable of testing the separated parts, but also must be able to perform integration tests. In the last few years, an enormous amount of effort was invested into the testing environments. All the main development languages have advanced testing frameworks. To mention some of the biggest ones, I should name Selenium [38], Arquilian[5], Cucumber[9] and Autotest[6].

The current extremely competitive market of electronics of all kinds is very sensitive to the time it takes to introduce new products. Errors in design and implementation of a product, not only increase the cost of the final solution, but also cause delays that, in the end, mean a financial loss.

This drives the demand for fast and efficient testing systems. These testing systems must tackle several challenges:

- to provide a high level of automation of the testing procedure,
- to restrict the time needed to discover an error, this includes a fast rebuild of all tools that are needed for testing,
- to clearly identify an error and provide adequate information about the error,
- to define clear metrics to measure the progress of the testing process.

This thesis is going to discuss the area of testing hardware software codesign [11]. The hardware software codesign deals with the design of new embedded systems. Such a kind of systems can be found in a wide variety of devices, such as network routers or printers.

Embedded systems consist of one or more application specific processors (ASIPs). Each processor usually takes care of a single specific task and is, therefore, highly optimized for this task. The optimization is also the main difference from general purpose processors, such as the x86 family, which have to take care of various tasks.

The production of ASIPs in 2015 formed over 98% of the overall processor production. Therefore, this area is extremely important. Technology used for the creation of any ASIP is called System on the Chip (SoC) [37]. Such a technology allows integration of several ASIPs on one chip together with peripherals, such as memories, busses and others.

The development of current ASIPs must be done in a very short period of time [41]. In order to do so, it is common to use tools for the hardware software codesign. A hardware description language (HDL) is allways in the core of such tools. The development is done in a modern integrated development environment (IDE) that allows the designer to generate all the necessary tools, such as a compiler, an assembler or a simulator [35]. Then it is common that the application can be compiled in the same environment and simulated. These tools enable the Electronic Design Automation (EDA) and sometimes are also called the EDA tools [42]. Into the category of EDA tools falls, for example, the Processor Designer [40].

In the case of such a complex tool as the hardware software codesing environment, the testing techniques should be very advanced and ensure thorough tests of separate components as well as integration tests. In this thesis I will focus mainly on testing of the toolchain and particularly on tests of the compiler, as the compiler plays a key role in programming of an ASIP.

The thesis is divided into eight chapters. The chapters are organized in the following way.

The second chapter is called State of art. It describes the standard C language library, together with the testsuites that are used for compiler testing and the continuous integration systems.

The third chapter describes the Lissom project. It is targeted at the description of the toolchain, the software development kit (SDK), the way it is generated from the description in the ADL.

The fourth chapter is devoted to the porting of the library. It describes the role of the library in the toolchain, the process of porting and also automation of the porting process.

The fifth chapter discusses the problems connected to the scheme of test selection. As I use tests from a large number of sources, I need to deploy an efficient test selection mechanism. In the chapter I describe such a method and also the way how to automatically generate files that take care of test selection.

Chapter six focuses on the area of testing via a continuous integration server and also acceleration of such testing. This chapter introduces an improvement in the flow of testing jobs that brings significant time and space savings.

Chapter number seven is the last of the sections that are focused on the solution of testing problems. It deals with problems connected to the generation of testing jobs, describes the design and implementation of the generator of the jobs.

Chapter eight concludes the thesis. It gives the summary of the results, describes the utilization in the industry, the advantages and disadvantages of the chosen solution. At the very end of the thesis, the future work is also discussed.

Chapter 2

State of art

In this chapter I will discuss several areas that are connected with the standard library, testing of the compilers and the infrastructure that is needed for testing of the tools for hardware software codesign.

2.1 Standard library

The language, whose compiler is generated, is based on the grammar that defines the syntax of the language. But the compiler itself is difficult to use. What makes the compiler really useful is the standard library of the language, whose compiler is generated.

That is true for majority of programming languages. Because I am interested in the C programming language, I will have a look at the library of the C. The library for the C language is specified in the standard [3]. It is the subset of the C library POSIX specification. It is also called ISO C library.

In comparison to standard libraries of other languages, such as Python, the standard library of C is small. It provides only the basic sets of mathematical functions, functions for the conversion of types, basic manipulation for strings and file and console-based I/O.

When I compare the library with other language libraries, such as C++, Java or even Python, I find that it really holds just the minimum of functionality. Other language libraries provide, for example, containers, GUI tool kits or networking tools. The exact opposite of the C standard library is the Python standard library. The Python standard library provides, for example, clients and even servers for the common network services or multimedia services.

However, there is one big advantage of the small standard C library. It is the fact that in order to provide a working version of the library for a new platform, the amount of effort I need to expend is relatively small.

The main parts of the standard C library are the following:

- *Data types* - The data types provide the declaration of how the data are stored and what operations are permitted over the data.
- *Character classification* - In this section there are declared functions that are used for the test of the character membership, for example `isdigit()`.
- *Strings* - A set of functions that implements operations over the character or byte strings, such as a concatenation or a copy.

- *Mathematics* - An implementation of the basic mathematical functions for integer, float and other data types.
- *File input/output* - An implementation of many functions for the standard input and output. The function forms the main part of the `stdio.h`.
- *Date/time* - Functions that provide conversions between the date and time formats, a time acquisition.
- *Localization* - An implementation of the basic localization routines.
- *Memory allocation* - Dynamic memory allocation, the heart of the library, functions like `malloc, realloc`.
- *Process control* - A very important part of the library, basic functions for starting and termination of the process.
- *Signals* - Closely related to the process control, definition of the program behaviour when it receives the signal.

Some parts of the library are more error prone than others. There are certain parts of the library that are well known for overflows, such as `gets()`, and some of them are deprecated. Other functions are considered *thread unsafe*. None of these are crucial for the developers as there are always ways how to overcome such problems.

Even though there are several different standard C library implementations, the above mentioned parts are common for all of them. I will now have a closer look at the *Newlib library* as it plays an important role in the thesis.

Newlib Library

The Newlib library [33] is a collection of several parts that are all distributed under free licenses. It is the C standard library implementation that is intended for use in embedded systems.

The library is currently maintained by the Red Hat corporation [22]. The Newlib project is currently used in the majority of commercial and also non commercial embedded systems. It is particularly popular for the ones without an operating system.

The library has a strong support for porting (an addition to the new platform) and because of its popularity, there is a lot of documentation about the porting, for example [19], [7].

It is very well prepared for the addition of a new platform. It is divided into two parts. The first one is the `newlib` directory. It contains the majority of the code for the two main libraries `libc` (the core of C library) and `libm` (the mathematical library). Some architecture specific code might be found here.

The other part is the `libgloss` directory, called also `Board Support Package` (bsp), contains the platform dependent code. Therefore, during the porting mainly, the `libgloss` directory has to be targeted.

2.2 Test-suites for the C compiler

As my work deals mainly with the C compiler, I will focus on the sets of tests that are designed for the C/C++ compiler. The majority of the big compiler projects, such as GCC and LLVM, are distributed together with compiler test-suites. But there are commercial test-suites, such as the *ACE test-suite* or the *Perennial test-suite.* Companies developing such testing sets are very well aware of the fact that compiler testing is a growing area. The standard techniques are not able to cover the needs of the modern compiler development.

The test-suites are mainly used for *regression testing.* The aim of regression testing is to ensure that the software does not contain bugs, which we have uncovered during the process of development. The GCC test-suite and also the LLVM regression tests are sets of tests written by the developers of the compilers. The bugs were either found by the authors or were reported by the users. By execution of this test set I ensure, that the already known errors do not reappear. But by this approach I am not able to discover new errors. Very seldom do the already written tests trigger a new unknown sequence that results in an error.

2.2.1 GCC test-suite

The GCC test-suite [20] is a part of the compiler from the early stages of the development. It is distributed under the same licence as the compiler and contains a vast number of tests, which is true for all the other test-suites as well. The GCC test-suite does not come with the infrastructure and has clear reports, once the testing is finished.

The test-suite contains various types of tests. There are tests for C as well as for C++. As we do not support the full C++ in our project, I use mainly the C tests for the testing. There are very simple programs, as well as larger programs, such as *SHA* or *Dhrystone algorithm.* The tests are very well sorted into directories. One of the greatest disadvantages of the GCC test-suite is the fact that the tests are not sorted. There is a certain directory structure, but it is very vague. For example, if a user wants to filter the float tests or tests that use only integers, they must do it by themselves.

The test-suite contains the torture part. These tests are meant to be compiled several times with different options. The torture test-suite is divided into several directories. Some tests are designed to be executed after the compilation but there are also tests that are designed only for compilation and should not be executed.

The disadvantages of this test-suite are very similar to the disadvantages of the compiler. The project of the GCC compiler is quite old and so is the test-suite. Moreover, the tests are usually only added to the test-suite. There are test cases that once triggered an error in the original code, but the code is no longer part of the compiler. Another problem is the fact that tests are not properly sorted and the test-suite does not contain an infrastructure. Although this can be viewed as an advantage, as I do not have to modify the existing code.

2.2.2 LLVM test-suite

From the LLVM project [29] there also comes a test-suite. This test-suite has two major parts. There is a regression test-suite and the benchmarks.

The regression test-suite is similar to the GCC one, which was described above. This part contains the test cases gathered during the development phases. The test cases are usually small pieces of code, which test a specific feature of the LLVM or trigger a specific bug. The language they are written in depends on what part of the LLVM is tested. The test-suite possesses a special driver for such tests, it is called *lit.* The directory, which contains the regression tests, is further broken into subdirectories that are named after the parts of the LLVM compiler that are tested by the cases contained in the given directory.

The other part of the LLVM test-suite, which in this case means benchmarks, is very different from the GCC test-suite. The LLVM test-suite is in fact composed of various benchmarks. The smaller programs meant for regression are kept separated. The rest of the test-suite, the benchmarks, are sorted into directories and thanks to the well designed makefile system the user can easily enable and disable the directories.

The system for the benchmark compilation is hierarchical. There is a system of makefiles which control the compilation as well as the execution of the benchmarks. Each benchmark can, therefore, be compiled and executed separately.

The system enables a parallel compilation and execution of the benchmarks, which keeps the speed of the testing at a very good level. The system is able to detect the number of cores and run the compilation and execution on several cores. However, due to the number and complexity of the test and also the fact that the tests run on a simulator, the testing is slower then I would expect.

When I look at the mechanism for the test selection, it gives the user a possibility to modify the compilation and execution of the benchmarks at will. But what is missing is the possibility to choose the benchmarks according to some predefined features.

2.2.3 Test selection mechanism

One of the most important criteria for use of the test-suite is the way of test case selection. Both of the test-suites that were mentioned had serious drawbacks as far as the test selection is concerned. There are certain test-suites that do not possess any testing infrastructure and test selection mechanism at all. The rest of the test-suites gives just very basic options of the test selection.

The test selection is usually based on a simple list of files. In certain cases, the list of files contains only the test name, but in other cases, it contains the whole path to the test from the given directory, which is typically the root directory of the test-suite.

I need to focus on specific aspects of the test selection mechanism. An important role here is played by the information about the instruction set that the compiler possesses. Very often the model from which the compiler is generated can dispose of a specific bit width. For example, I can create a compiler for the 16-bit model or for the 32-bit model. This characteristic influences the set of tests that can be compiled and executed. There are also other factors, such as the presence of the C compiler library and the presence of compiler-rt library and so on. All of these factors must be taken into account.

My test selection mechanism must be able to address such differences. I need to easily choose the test for each platform according to the bit width and the presence of certain libraries. And, in certain cases, also to specify directly that certain tests should not be executed on the given architecture.

2.3 Continuous integration

The continuous integration servers are nowadays used for deployment and testing of new packages and releases. Before the continuous integration method was deployed, the development of software had had to deal with several serious disadvantages. The teams of developers merged the code together via non systematic methods and they were very often forced to rewrite certain parts of the code. A process like this very often took weeks and sometimes even months. This very often led to inevitable delays in the process of development [30].

Nowadays, we use modern tools for the process of software development, these make the whole process faster and easier. Because today the software development is not only the coding but also continuous testing, version control of the code, quality assurance and observation of metrics. Continuous integration tools make this process faster, less error prone and they also help with automation of certain parts. It gives the programmer a powerful tool for error detection and also reporting of errors, and it also helps with the release management.

The most widely used continuous integration and continuous deployment server is called *Jenkins* [25].

The Jenkins is an open source continuous integration server. It is implemented in the Java language. It has a very simple interface, which can be easily customised by a large number of plugins. The plugins can be divided into several categories:

- *Version control system plugins* - plugins that provide interface to the most common *Version Control Systems* (VCS),
- *Executor plugins* - plugins that allow execution of certain scripts, such as Python,
- *Metrics and visualisation plugins* - this group of plugins allows a visualisation and provides support for various kinds of results.

One of the biggest advantages of the Jenkins project is the speed of development. There are updates and bug fixes available every week. There is also a more stable version that is released three times a year. This version contains only packages and bug fixes that are considered stable.

2.3.1 Jenkins as a build environment

The Jenkins is nowadays widely used as a tool which performs nightly builds and tests. Let me introduce the most important steps of the build. The build is a job in the Jenkins that is configured in an appropriate way. I use two kinds of job for the build, the `multi-configuration` job and the `maven job`. The jobs differ just in the execution step, otherwise they are very similar.

The first important feature that can be configured is the job security. The job can be configured in a way that other users can just watch it or control it, etc. There are several plugins that modify the basic functionality of the Jenkins in this area.

A user can also set the names of the jobs that will be able to copy the artefacts in the configuration if the job stores any. Moreover, the job parameters can also be configured. In the Jenkins, there are basic kinds of parameters, such as boolean, string, text and new kinds are added by the various plugins.

Another extremely important part in the job configuration is the *Source Code Management.* All version control systems can be added into the Jenkins environment via plugins. Because we use the git version cotrol system I am most interested in the git possibilities in Jenkins. There are plugins for integration with git [21], such as Gitlab, Github and also GitBucket.

Then there are the sections *Build Triggers* and *Build Environment.* In these sections, the user can configure a periodical build. This is useful especially for nightly builds and tests. Also the polling can be configured there as well as other actions, such as execution. What is extremely useful is the build abortion. There are several possibilities, such as the *absolute timeout* or the *conditional timeout.* Also the environment variables can be set for injection into the job.

A very important part in the multi-configuration project is the *Configuration Matrix.* The most frequently used axis is the one containing nodes. The user can define what slaves will the build be performed on. It is possible to choose *Labels* or *Individual nodes*. Also another axis can be added, such as an axis based on a version of the Java language.

All the above mentioned sections can be considered a configuration. After these steps comes the build. The build is divided into the *Build* and the *Post-build actions*.

In the *Build* section, the user can configure an execution or a conditional step. From my experience, it is better to configure the execution and do the conditional steps inside the scripts. There is also a possibility of executing other projects before the execution starts. The kind of the offered executors is affected by the installed plugins.

The last part is called the *Post-build actions*. The possibilities offered here are wider than the ones in the *Buid Step*. It is possible to execute some clean up procedures and also wait for other projects until they finish the build. Very often, the job archives some artefacts and they can also be configured in this step, as well as the trigger of other jobs.

The job is stored in the xml format in the Jenkins. The extensions just bring the new marks into the existing jobs.

2.3.2 Current possibilities of the job generation

Let us have a look at the current development in the field of job generation. I can distinguish between two types of solutions. There are tools in the Jenkins that were designed for this purpose and then there are several works that try to deal with the problem of job generation outside of the Jenkins environment.

Another possibility provided by the Jenkins server itself is the *Job generator plugin* [2]. This plugin is based on the template, which is the job itself and the parameters, which can be global or local. This plugin is very powerful in combination with other plugins, such as plugin for the conditional resolution. However, it shows limitations in the form of what types of jobs can be generated and it cannot use time triggers. Moreover, it is very difficult to generate more complex jobs. The hierarchy and conditions can become very complex and the whole process is quite error prone. I also did not find a way how to set the desired nodes in the multi-configuration project.

The most powerful solution from the Jenkins itself is the *DSL plugin* [1]. The dsl plugin offers the possibility of definition of the job, which will serve as a template. From this template the Jenkins is able to generate other jobs. This is done via a special build step called `Process Job DSLs`. The build step executes the script in the Groovy language. This solution allows the user to perform basically any customization over the template. The Groovy language is very powerful. On the other hand, this solution is still within the Jenkins environment and can be affected by other plugins, which can cause problems. Moreover, the Groovy language is not very common and may require complicated settings.

I will introduce one approach that try to deal with job generation outside the Jenkins environment. Interesting ideas were proposed in the article at the Jenkins User Conference [27]. The article deals with the automation of testing in the area of robotics. The author uses combination of various Jenkins plugins for packaging and a static analysis. Nevertheless, the process of the building and testing is very complicated and hardly maintainable. The author of the article proposes the use of the Domain Specific Language (DSL) for the specification of information and then generation of the Jenkins jobs. It seems that the author just uses the Jenkins for the building. However, the system seems to be slow and problematic as far as the synchronisation of the jobs is concerned. Also there are problems with the graphical side of the solution.

Chapter 3

Lissom project

In this section, I will describe the Lissom research project [28], which creates the background for the testing methods that are described in this thesis.

The Lissom project has two main areas of interest. The first one is the ADL called CodAL, for the ASIP description. The description of the language can be found in detail here [31].

The second scope of the project is the generation of the full toolchain from the description in the ADL CodAL language.

3.1 CodAL Language

The CodAL language falls into the category of mixed ADLs. This means that the language is able to describe the architectural information needed for the generation of the C compiler and, at the same time, to provide information about micro-architecture, which is needed for the generation of the hardware.

The CodAL language is special for the fact that the description of the core is created in two levels of abstraction.

- instruction accurate,
- cycle accurate.

The first one, the *instruction accurate*, is on a higher level of abstraction. This description is very simple and it is written in a C-like code. It describes the instructions. The addition of the instructions is very straightforward and for an experienced user, it takes only several minutes to create the first version of the core with few instructions for which the basic tools, such as an assembler and simulator, can be created. The designer can fully focus on the instruction set without considering the complicated micro-architecture. From this level of description, the user is also able to generate the C compiler and the profiler.

The *cycle accurate* model is more complicated. On this level, the micro-architecture is described. Things, such as pipeline, hazards, etc. must be taken into account. This description is taken as a base for the synthesis. This level of abstraction gives the user a possibility to generate the description in the hardware description language, the functional verification environment, the simulator, the assembler and the profiler.

There is a large number of files that are common for both descriptions and these files are shared between the descriptions. There might be several equivalent descriptions on the cycle accurate level that correspond with one instruction accurate model. This is logical, as the instruction set must be the same, but there might be several hardware variants that are optimized for the speed or power consumption.

3.2 Toolchain

As I have mentioned before, the automatic generation of the full toolchain is one of the two main tasks of the Lissom project. The generated toolchain contains all tools known from other toolchains but it also contains specific tools.

The toolchain that is described below creates an entry point into the testing of the compiler. The generation itself is very often also a part of the testing. Moreover, the toolchain stands as a prerequisite for the tests of the compiler.

All the tools are generated from the description in the CodAL language. At the beginning, the model in the CodAL language is validated and compiled. The result of the compilation is the XML representation of the model.

Once the XML is created, there are two tools working over it. These tools are the toolchain generator, called also *toolsgen*, and the semantics extractor or *semextr*.

The toolchain generator produces tools, such as the simulator, the assembler, the debugger, profiler and so on. The tools that are generated by the toolchain generator consist of two types of files. Both types of files are compiled and linked together.

1. The files that are *platform independent* are the same for all architectures. Into this category fall user interfaces with parsers of the command line arguments, or in the case of aprofiler, the generation of the graphical output.

2. The files that are automatically generated, such files contain *platform dependent* information. Into this category fall the instruction decoders in the simulators or assembler printer in the C compiler.

The second tool is the *semantics extractor*. This tool was thoroughly described in the dissertation thesis [24]. The semantics extractor is the prerequisite for the compiler generation and also decompiler that is described in the thesis [26].

The extraction of the semantics is possible only from the instruction accurate model. The extraction from the cycle accurate model is not supported. The information for the semantics extractor is contained in the suitable form only in the instruction accurate model. Therefore, if the user wants to get the toolchain together with the hardware it is necessary to create instruction accurate as well as the cycle accurate model.

Once the file with the extracted semantics is created, it is used by a tool called *backend generator*. This tool creates the only platform dependent part of the C-compiler, the backend. The rest of the compiler, the frontend in this case the Clang and middleend, the optimizer are platform independent. The backend part of the compiler uses the information from the semantics extraction for pattern based matching for the most suitable instruction.

The complete toolchain can be generated from the description in the CodAL language. The exported toolchain can be stored in a specific directory structure that contains the tools together with the libraries that are needed for the execution.

Chapter 4

Porting of the C library

The first part, which is needed for automatic compiler testing of processors for embedded systems, is the support of the Newlib library [12], [18],[16]. The variety of programs that can be created without the support of the standard C library is very limited. Therefore, the availability of the library is crucial and its position in the process of testing is unsubstitutable. I have worked on the first version of the Newlib port that will be described here.

4.1 Theory of Porting

The main reason for porting the library on the new platform is the fact that I need to add support for the call of the C functions. To be precise, I want to use the *libc* functions, such as `printf, malloc, free`, etc. in programs that will be used for testing of the compiler. And because I do not possess the development kits for all the platforms, I use simulators instead. Therefore, I must add the new platform into the Newlib library and our simulators must know how to deal with the Newlib library calls. If one does not grant libc library support in the simulated environment, the number of constructions which can be used and tested is very limited. Consider the following simple example written in C:

```
int main(int argc, char **argv)
{
    if(strcmp("alpha","beta")==0)
{ return 1;}
    else
{ return 0;}
}
```

Even this simple program can hardly be executed because it uses the function `strcmp` that is part of the standard C language library. This program cannot be compiled, unless the file of `string.h` is included and a possibly some other header files are included also.

On the contrary, the main aim of the testing process is to cover as wide area as possible and also to try as many different combinations of the function calls as we can. However, this goes against the idea of embedded solutions, which are usually specialised in just one single area. Furthermore, because I focus especially on the embedded systems, I do not even try to cover all the functions provided by the standard C language library, which is in my case the Newlib. In fact I will use and therefore test only those functions that can run under the simulated environment and are useful for the programs that will be executed on the given platform. Moreover, the embedded systems are not designed for the use of the vast number of constructions that the programming languages offer these days. Typically there is just one task, usually quite a complicated task, which is launched repeatedly. However, during the design of the chip it is often unclear what part of the library will be needed, so I will have to port the whole library and reduce the size later if it is necessary. There are certain areas that are more likely to be removed from the library than others, for example:

- *threads* - I assume that in simple programs for embedded systems one will not use threads.
- *locales* - All the locales were removed from the library.
- *inet module* - Even though networking plays an important part in modern embedded systems, in some cases the module can be disabled.
- *files and operations with files* - Certain simple application do not need interface for working with files.

Now I will introduce important parts of the library. Simply said, all that really has to remain from the library are the `sysdeps`. The `sysdeps` are the core of the whole system (how to allocate more memory, etc.), then important modules, such as `stdio`, which takes care of the outputs and inputs, and other modules I wish to preserve. In this case, I wished to preserve the following parts of the Newlib library:

- *stdio* - This is one of the main reasons for porting the library, which is to get in human readable form output from the simulator.
- *module for strings and memory* - In many applications I would like to use functions, such as memcpy, strcpy, strcat, etc.
- *memory functions* - For example `malloc, free, realloc,`
- *abort and exit.*
- *wchar support* - But without the support of different encodings.

Let us have a look at the functions that remain in the library. The functions can be divided into two groups. The first group consists of functions that are completely serviced within the simulated environment. For example, the function `strcmp` falls into this category. This function and its declaration remain unchanged within the simulator if they are written in the C language that does not require any changes. These functions are not tied to a kernel header files, so there is no need to change them.

The second group of functions consists of functions that are translated to the call of system function. The function `printf` can be used as an example of this group of functions. The call of `printf` function can be divided into three phases that are illustrated in the following picture 4.1.

At the beginning, the call of the `printf` function is translated to the call of a system function, with the highest probability it is going to be the call of the function `write`. Write is the function call, that is serviced by the operation system, and hence is system dependent. But as I want to use the simulator on the UNIX platform, as well as on the Windows systems, I have to get rid of these dependencies. To do so, I will use the special instruction principle.

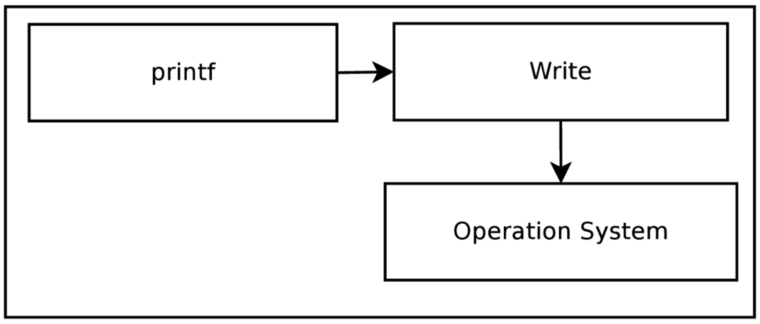

Figure 4.1: Scheme of the printf function call

4.1.1 Special instruction principle

The special instruction principle means that I will use an instruction with the *OPeration CODE*, *opcode* that is not used within the instruction set for a special purpose. So far all architectures that were modelled within our research project had several free *opcodes*. It is typical that the instruction sets do not use all operation codes which are provided. But in the case of no free opcode, this method cannot be used. The special instruction principle will be used for ousting the dependencies on the kernel header files.

Functions provided by the operation system are triggered by the *syscall mechanism*. The system calls can be quite easily detected. Each library should have defined the syscall mechanism in a special source file. This syscall mechanism differs, as they usually are platform dependent. So i386 architecture will have a different syscall mechanism from the ARM [4].

The syscall mechanism is in fact a wrapper. The call will be passed to the simulator that will do the call and return the result.

4.1.2 Simulators

As was described above, all simulators are generated automatically. At the beginning, the source files are generated by specialized tools. When the generation phase is finished, the simulator is build by the `Makefile` from the automatically generated files and also from the static files. It will be necessary to add the following information into this process:

- Information about which instruction calls the system function.
- The simulator will have to know the convention for storing parameters.
- The simulator will have to recognize which system function is going to be called.
- The simulator will have to perform the call of the correct system function.

The first three points will be solved within the model of an instruction set. The instruction with the opcode that is not used will be declared. The instruction behaviour will be defined in the following way: according to the parameters it will call the given system function. The simulator will have to recognize the system it runs under, and call the correct function. For example, on the UNIX system it will be the function `write` and in the Windows the `WriteFile`. This problem should be solved by the `libc` library of the given platform.

The parameters that were placed at the given position at the simulated memory can remain unchanged. They will be later passed to the specific system call.

4.2 Automation of the porting process

By default, the Newlib uses the system of make. I have put quite a lot of effort into the automation of the whole process [13]. The modifications were made to the Newlib library, so it now uses the CMake system. It was divided into two parts that are placed in separate directories. One part is common for all platforms. This part is placed in the directory called the `newlib`. The directories that contain platform dependent files are stored in the directory with the model. This is done in order to have all the platform dependent files in one place in the strictly given directory structure.

Let us have a look at the platform dependent files. Strictly spoken, the directories do not contain only platform dependent files. There are also files that are the same for all the platforms but the division is done on the level of directories and not on the level of the files themselves. The directories that are kept together with the model are the directories `libgloss` and the directory `newlib`, this is the subdirectory of the directory `newlib` mentioned the paragraph above.

While the directory `newlib` contains mainly header files with various settings and definition of the `setjmp.S`, the directory `libgloss` takes care of the syscalls handling. The syscalls are very important for our project because this mechanism allows us to get the information in and out of the simulator. I will focus on the way how to automatize the process of syscalls creation.

There are several ways how to cope with the syscalls porting. After I gathered all the necessary information about what syscalls are necessary for the simulation and tried several ways of implementation, I found out that only a very small part of the syscalls must be written in the assembly language. The rest can be written in the C language and that makes the code platform independent. The Newlib defines 20 syscalls but I need just 6 of them.

Nevertheless, the rest of the syscalls could be implemented in the same way as the six supported ones. The syscalls are defined in the header file and have numbers from 1 to 20. The first six are the supported ones and the rest of the numbers is assigned to the unsupported ones.

For the syscalls themselves, I have defined the structure called `params`. This structure contains the parameters that are needed for each syscall. This structure slightly varies depending on the actual syscall. But it is written in the C, which makes it also platform independent. What is only written in the assembly language and is, therefore, platform dependent is the `PERFORM_SYSCALL` function. In fact it is not a function but a multiple line macro defined in the *inline assembler*. Let us assume that a multiple line macro can have the following form:

```
define PERFORM_SYSCALL(ADDR) \
   __asm__( "REGr1=add REG0,%0" : :"r"(ADDR)); \
   __asm__( "syscall");
```

This macro is not taken from any existing processor. I have defined it just for the model purpose. Now let us have a closer look at the macro itself. This macro takes only one parameter. The `ADDR` parameter is the address of the structure that contains the parameters of the syscall as mentioned above. This address is assigned to the register that is used for passing of the parameters. This register can be specially marked as it is often used for passing of parameters. Then there is the special syscall instruction, in this case it has the name `syscall`. These two lines can be determined from the description of the core performed in the CodAL language. I will propose a way how to create the macro semi-automatically. Consider that the `PERFORM_SYSCALL` macro itself is a template. The necessary information can be filled into this generic template before the compilation time of the library. First let us have a look at the syscall instruction. I simply scan the model for the instruction that bears this name. If the instruction is not found, I search the model for the construction in the following form: When this construction is found, I use this instruction in the second line of the multiple line macro. Please note that in this case, the instruction does not take any parameters. If this instruction was parameterized, I would determine the parameters from the syntax. Nevertheless, this instruction does not have to be found. In such a case, the template would be incomplete and an error should be reported. The process is shown in Fig. 4.2.

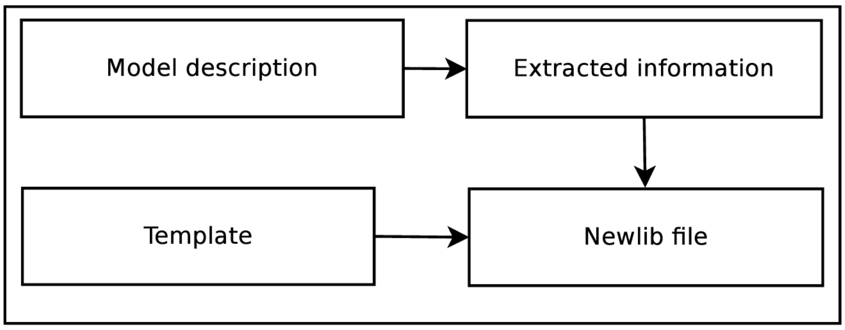

Figure 4.2: Scheme of Newlib file generation

As far as the first line of the macro is concerned, I need to assure that in the register, which is used for passing the parameters, I assign the address of the structure with the parameters. So I search the model for the instruction `add` or instruction with similar functionality. In the syntax section of the instruction, I find the actual form. Then I find the register for passing parameters in the model that also bears special description. From these parts of the information, I should be able to put together the first line of the macro. This approach works for standard architectures. But there may occur architectures for which there might arise difficulties. The Newlib library, in the current version, supports only 32-bit architectures.

4.3 Experimental results and contribution

For having a comparison with commercial compilers, I tested the automatically generated compiler with the commercial Perennial test-suite. The results described here were gained from the generated *MIPS* and *Codasip uRISC* compiler. The testing was performed on a complete toolchain. The tests were compiled by the generated compiler and afterwards executed the tests on the simulator which was also automatically generated by the tools from our project. I have only a part of the Perennial test-suite. I used only tests that examine the core of the compiler. I excluded some of the tests that cannot be compiled due to the header files dependencies, which I do not support. The tests in the test-suite are divided into groups according to the chapter of the standard that is tested. I use tests for the clauses 5 and 6. I have mainly tests for the standard C90 and several tests for C99 standard. The results are summed up in Table 4.1.

Core	*Tests without C library*	*Tests with C library*
MIPS	797	1680
Codasip uRISC	804	1688

Table 4.1. Comparison of number of tests.

In Table 4.2, I present the testing results with and without the presence of the C library. It is apparent that not only the number of tests is lower without the library but also the number of failing tests is very small. The presence of the library provides a better opportunity for debugging of the code and triggers more errors.

Core	*Failing Tests without C library*	*Failing Tests with C library*
MIPS	2	19
Codasip uRISC	0	8

Table 4.2. Comparison of failing tests.

The solution also brings a higher level of automation into the testing of the automatically generated compiler. I have introduced methods that simplify the porting of the library to the newly developed cores.

Amongst the biggest contribution I can place the following things:

- *enlargement of the number of tests* - Without the support of the C library, it is possible to test only a very limited set of tests, in my case the number of tests was increased three times.
- *speed-up of porting* - The library was rewritten in a way that it enables far faster porting for new cores, the number of codes which have to be written by hand has been significantly reduced.
- *higher level of automation* - The code that is common for the majority of the cores was introduced, as well as additional scripts for build automation and creation of the library, providing a higher level of automation than before.
- *larger number of failing tests* - It is often very difficult to trigger bugs without the support of the library, so it enables better test coverage and triggers a larger amount of errors that help to keep the compiler in a good shape.

The porting of the Newlib library and topics connected to the porting were published in the articles [12], [18],[16]. The articles describe the process of porting and its automation together with the results.

Chapter 5

Tests selection

As was mentioned in the section which discussed the test-suites, one of the weakest points, which does not suit my needs, is the test selection mechanism. I have decided to create a test selection mechanism that suits the needs of the testing system for the hardware software codesign [16]. It will form the content of the following chapter.

5.1 Test selection scheme

The test selection scheme that would be suitable for use in our project must fulfil several criteria. First of all, it must be independent of the source of the test, so it will be applicable for as large a number of tests as possible. It also must be robust enough and lightweight at the same time, so it should be simple to modify the tests I already have and addition of new tests must not be difficult. It should not only work for tests from the regression test-suites, but should also be applicable to tests from random generators.

5.1.1 Test selection phase

As I have a large amount of tests from different sources, I need a universal approach that will define which tests are suitable for compilation and execution on the given platform.

I have created a system of files, which restricts the number of tests that can be compiled on the given platform, based on the libraries that are available. The libraries are just one of the test selection criteria. Other characteristics are also taken into account, for example, the size of the registers or the size of the stack.

Currently supported features which can be used for the test or directory selection are:

- *architecture* - Certain tests or directories can be disabled for the given architecture.
- *libraries* - Tests can be disabled if a certain library is not present.
- *bit width* - Test selection according to the bit width.
- *level of description* - Often some tests, containing system calls, cannot be used for a cycle accurate model.
- *purpose of compilation* - Some directories are disabled, for example, for functional verification.

The naming convention for the files, which are used for the test selection, is very simple. The file bears the same name as the test does but it has the suffix `.x`, instead of `.c` or any other. The system is a hierarchical one. It is possible to have a hierarchy because I support nesting of the directories and I keep the `.x` files not just for the tests, but also for the directories. In the case of directory, the selection file has the same name as the directory with the `.x` suffix.

These files possess as minimal functionality as possible. I try to keep their size minimal. The typical functionality of the file is that, based on the value of the flags, the test is excluded from testing. I should say that implicitly all the directories and all the tests are selected for testing. So, if I want to exclude the tests, or whole directories from testing, I have to indicate this.

As the size of the files is kept minimal, the functionality and flag settings must be done in another place. This functionality is kept in the main testing module. The functions that check the current state of the flags and control what libraries are accessible for the linking to the given platform are declared here. The centralization has a purely practical base in this case. The typical usage of the `.x` files is that I disable testing of the whole directories according to the libraries that are accessible. The `.x` files can also bear other functionality. It is possible, for example, to set different variables. I can specify flags that should be added to the compilation or add some files to the linker as in the following example.

```
if [ "$C_LIB" == "0" ]; then
     FILE_DEPS+=crt0.o
fi
```

On the level of files, I most often use the `.x` files for filtering the tests that depend on compiler-rt library for the given platform. The compiler-rt library provides software implementation of the float and double operations. Usually only a few tests in the given directory depend on compiler-rt and the dependence does not have to be the same for all platforms, the best solution is to condition the test execution by the platform and compiler-rt presence. This is demonstrated in the following example.

```
is_arch "mips_basic" $1
       if [ "$?" == "0" ]; then
         if [ "$RUNTIME_LIB" == "0" ]; then
            RUN_TEST=0
        fi
      fi
```

The biggest advantage of this approach, and also the main reason for introduction of this system, is its universality. I deploy the tests from the llvm test-suite [29], gcc test-suite[20], Mibench [32] set of tests and I also have tests that were created within our project, and I have also generated tests. The system of the `.x` files can be used for all these sources, as long as I use just the tests without the testing infrastructure that is provided in several cases. The only set of tests, which I tried to use together with the infrastructure that is provided together with the tests, is the Perennial test-suite [34]. After several iterations, I have also started to use the Perrenial tests with my infrastructure for the tests execution.

5.1.2 Test compilation and execution

The compilation of tests is performed in the central module. As I have the system of the `.x` files, I enter only those directories that I know are suitable for testing on the given platform.

So, before I enter a directory with tests, I check the `.x` file for the given source and consult the restrictions that are defined by the `.x` file and set all the variables denoted by the file.

If the directory is feasible for testing, I cycle through the tests in the order denoted by the test list. The `.x` file is always checked first, and if nothing blocks the procedure of testing, the test is compiled. The presence of the `.x` files is not compulsory. As mentioned above, the default setting is to cycle through all the directories and execute all the tests. However, if the file is present, it will be checked. When the restrictions are not met, the file is skipped.

Should there be any problems during the test compilation, they are logged. I log the standard output as well as the error output. I keep a list of tests that were not compiled successfully together with the output of the compiler. The logs are kept for every platform that is tested to avoid overwriting. It is also possible to create a unique log not just for each platform but for every run of the testing system. These logs could be, in the future, stored in the database to keep precise testing history.

5.1.3 Logging information and test evaluation

The test evaluation is kept decentralized. Because I deploy tests from different sources, I need to keep the scripts that provide the test evaluation together with the tests. Some tests are evaluated on the basis of the exit code, but there are tests that produce, for example, the text output and I have to compare the output with referential values. In these cases, the `Newlib` library is used.

As in the case of test compilation, I keep detailed logging information. I keep the output of the simulator and after the test evaluation I put it into the list of passed tests or failed tests according to the result of the evaluation. The logs are created for every tested platform and can bear the time reference.

The results are kept in two different files. I log the successful and unsuccessful tests in two independent files. The files are created for every directory that is tested. Each file with the results has a special header, which stores data necessary for the test archiving.

5.2 Generator of the test selection files

The mechanism that is explained above has met the needs of our research project. However, as in our project we very often add new models and branches that need to be tested, we also need a way how to easily create a new file, that modifies the test usage, or to modify the files that already exist.

The best way for doing so, is to create a generator of such files. The generator would need the information about the tested platform as well as about the tests themselves. It would also very nicely fit into my plans about the high level of automation of the testing process. In the following subsection I will introduce such a generator.

5.2.1 Design of the generator of test selection files

The main task of the generator will be the creation of new `.x` files and also update of the existing ones. The generator will need the information about the platform that includes mainly:

- *bit width* - Is the platform 16/32-bit or does it have a different size?

- *availability of the libraries* - Do we have a compiler-rt library or any other library for the given model?
- *availability of instruction and cycle accurate description* - What level of description do I possess?

This is the main piece of information which I need to get about the platform. The majority of such information can be easily gathered. I will have a look at various possibilities in the implementation part of the generator.

The knowledge that I need to have from the side of the tests is a little bit less complicated. I just need to know what header files the test includes. I can say that if the test includes any header file, such as the test below, I need to generate a corresponding file. The test below will require the presence of the `Newlib`, as well as the presence of the `compiler-rt`.

```
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <double.h>

double res(float i, double j){
    double res;
    res = M_PI*i*i*j;
    return res;

}

int main(){
    float i = 3.14159;
    double j = 4.9685;

    double res = mul(i,j);

    printf("%d", res);
    exit(0);
}
```

But the situation is not that straightforward. Certain tests might rely on availability of the library and not include any header files. Moreover, modern compilers in such situations do not exit with the error code, but just emit a warning and compile the test if the header file is available.

It seems that the only proper way how to find out if the test needs the support of any library for the given architecture is to compile the file and to find the necessary information from the temporary files.

The information I need can be obtained from several sources. The best one is to get the desired information from the object file.

The most desired information is if there are undefined symbols in the currently compiled module. This information can be obtained via tools, such as *objdump*. Below there is an example of the object dump output with given parameters.

```
addvdi3_test.o:     file format elf64-mips_basic

SYMBOL TABLE:
g_str   000000000000  info_string10_addvdi3_test.s
...
```

```
00000000000000d0 l        .text  000000000000 tmp15_addvdi3_test.s
00000000000001ac l        .text  000000000000 tmp27_addvdi3_test.s
0000000000000208 l        .text  000000000000 tmp33_addvdi3_test.s
000000000000031c l        .text  000000000000 tmp53_addvdi3_test.s
000000000000037c l        .text  000000000000 tmp60_addvdi3_test.s
0000000000000000          *UND*  000000000000 __addvdi3
00000000000000c8 g      F .text  000000000338 main
0000000000000000          *UND*  000000000000 printf
0000000000000000 g      F .text  0000000000c8 test__addvdi3
```

From the description I can easily identify the undefined symbols, which in this case are `__addvdi3` and `printf`. This indicates that I will have to link the compiler-rt library together with the standard C language library.

I have shortened the example as it was quite long and it would not fit the page. Some irrelevant symbols and information has been left out.

Once I have the needed information about symbols and what libraries should be linked, I need to generate a new file or update the existing one. This should not be a difficult task. For the implementation I have chosen the Python language.

I have called the tool for the generation of the `.x` file the *Constraintgen*. The implementation of the tool was performed in the Python language and the framework *pytest* [23].

One of the main advantages of the pytest is that it collects all the files with the prefix or suffix test and executes them. It also uses the system of fixtures [36], which is a system of dependencies. These dependencies create a hierarchy that is resolved by the pytest framework.

For the implementation, I had to create a set of fixtures. The fixtures are responsible for the generation of the file, creation of the toolchain that is able to compile the source file and the compilation of the source file to the object format.

Once a single test file is compiled, the object format generator fixture parses the object file and resolves dependencies. After the resolution is finished, the resulted constraint file is generated. There are also other fixtures, such as the reporter or the model, but these fixtures play a subsequent role.

The inputs of the system are the directory with the model in the ADL language CodAL and the directory which contains the test, for which the `.x` files should be generated. This offers a possibility to create yet another layer above the *Constraintgen* that would offer an even higher level of automation.

5.3 Experimental results and contribution

With the implementation of the test selection generator *Constraintgen*, I have performed several tests. In Table 5.1, I have summarised a number of generated files for the MIPS and the Codasip uRISC core.

Core	*Number of generated files*	*Number of folders with tests*	*Time of generation*
MIPS	392	9	84.11s
Codasip uRISC	364	9	77.64s

Table 5.1. Speed of the generation

From the table, it is apparent that the number of tests is equal for both cores and the number of generated `.x` files is also comparable. The difference in the number of generated files is given by the fact that, in some cases, the compiler generates the call of the compiler-rt function while for the other core the call in not necessary. In both cases the majority of the files was generated because of the compiler-rt. The number of tests that required the Newlib library was lower.

The Table 5.1 also shows the speed of the generation. We can see that the speed of the generation is very good. The speed of the generator is approximately 5 `.x` files per second, which I consider very good. Should the `.x` file be created by hand, it would take approximately 10 seconds for the creation of a single file.

The major contributions of the selected solution are as follows:

- *flexibility* - The tests from various test-suites are supported, there is no dependency on the test source, so this system can be used for simple tests as well as for benchmarks.
- *higher level of automation* - The files that are used during the test selection are generated fully automatically without a user interference.
- *scalability* - The system can be used for any new core, the generator is able to gather all the necessary information from the compiler automatically.
- *acceleration of the testing* - The tool is able to generate the files fast.

The system of the `.x` files, which can be used for the test selection was published in the journal article [16]. The article sketches the scheme of the files.

Chapter 6

Acceleration of testing

In this chapter, I will discuss the speed of the testing. As was mentioned at the beginning of the thesis, there is a big pressure for deployment of new builds more than once a day. I will focus mainly on the acceleration of the testing [14], [17] as the build acceleration was the focus of the thesis by Lukasova [30] that I supervised.

6.1 Testing attitudes

The testing of various parts of the project is very time consuming. I perform various types of tests that have different time demands. I have spent some time by reorganization of the tests and investigating whether I can utilize the results between the various tests.

6.1.1 Testing oriented on tools

In the tools oriented testing, we need to ensure that the generated tools as well as the generators themselves work properly. So both these parts need to be tested thoroughly. There are also interesting interconnections between the generators and the generated tools that can save a lot of computer time.

Let us have a look at the generators first. The generators are in our case triggered via a command line interface. I have created a set of classes that enable us to perform full tests of the command line functionality in the Python language. This test-suite, in combination with various models, gives us a very strong tool for ensuring that our generators are stable. The test-suite is highly modifiable. I can also very easily enhance this test-suite with performance tests and stress tests. The test-suite can be executed in a mode which tests all combinations of the parameters that are legal. However, this is very time consuming and I often test only certain combinations of parameters. The results of the generators testing is one of the inputs into the testing of the generated tools.

When I get to testing of the specific generated tool, I first have a look at the tests of the generators. If I find out any problems during the generation, I either skip the tests as a whole or I need to pay more attention to the results of the testing.

If there have been issues with generation on all platforms, I skip the whole process of testing. If the tool has been generated correctly, I put the generated binary under tests.

Let's have a look at testing of the compiler `backend`. The input of the `backend` are the files that are in a certain kind of internal representation of the compiler driver and the output is the assembly code. Here it is possible to see the very close interconnection with the assembler, which is responsible for transformation of the assembly language to the object files. I have several ways of testing the compiler `backend`. The first line consists of simple tests taken from various test-suites, such as the GCC torture test-suite. These simple tests are meant for fast debugging of the `backend`.

There I can also utilize the results of the generators testing. Not only that I have to check that the `backend` together with the compiler driver were generated, but I can also check if the necessary libraries, which are needed by the compiler, are available. If not, I can choose only the subset of tests and shorten the testing time. If I do not have the Newlib library compiled, I can save up to several hours of testing. The time savings are also achieved thanks to the test selection mechanism, which allows automatic detection of the libraries.

The second line consists of benchmarks. The purpose of these tests is to tune the performance of the compiler. They can also be used for the debugging, but it is not as comfortable as in the case of the simple programs mentioned above. What is important in this case is the fact that I very closely observe the number of cycles that are needed for each benchmark. If I have a rapid growth in the number of cycles, it indicates severe issues in the compiler and can lead to increased power consumption, which is unwanted in the cores for embedded systems.

The last set of compiler tests are really complex tests, such as the Linux core. This category serves as the ultimate test that the compiler, as well as the model, contains the minimum of errors. The results of the generators testing comes to use in this case as well. In addition to all the tools that are required for the tests of simple programs, I also require the presence of the Newlib library. For execution of all three categories of the programs a simulator is used.

I have introduced a scheme of the utilization of the tools generator results on the compiler. Nevertheless, I think that it will give us the biggest time savings in the case of verifications. The reason for this is the fact that there is a large number of verification tests and they are time consuming.

6.1.2 Testing oriented on models

Another point of view of the testing system is from the angle of the models. The model developer expects that the tools work without problems. They are interested in their processor design and need to get the results of testing all in one place. Therefore, their use case is completely different.

The most model oriented tests, which I currently deploy, cover the area of functional verification. The role of functional verification is to verify the equivalence of the instruction accurate (IA) and cycle accurate (CA) model, which were described above. There are also formal methods [8], but they are not currently used in our project. The IA model describes the controller on the level of instructions, while the CA model is more precise. It describes a set of operations that represents the separate actions between the clock cycles. From each description a tool is generated. In the case of IA, I generate the simulator, and in the case of CA, I use the generated verification environment. I execute the same program on both and then I compare the results. Such tests are performed when both model descriptions are stable as it uses tools from the IA and CA description. These tests help us to discover differences in model descriptions.

One of the drawbacks of this attitude is the time demand. The test environment, which is generated from the CA description, is very slow and the number of tests is vast. It is not uncommon for these tests to take more than 24 hours.

Nevertheless, here I can also utilize the knowledge I have from the testing fo generators. Moreover, I need the results from the compiler testing as I use the compiled binaries for execution.

6.2 Case study and experimental results

I will demonstrate the whole process on the tools generator and tests of generated tools for one of the cores. The whole process is triggered by the nightly build. The job that is responsible for the nightly build is called simply `Build-Framework`. This job, once it is finished, triggers the job which is called `Toolchain-generator-codasip_urisc`. This job is responsible for performing tests of the generators. It performs all the necessary tests and produces a file with results in the form - test name: `result`. Should I have a set of tests with the names `fu-systemc, fu-verilog, fve-vhdl, fve-systemverilog`, the file with the results would have the following content:

```
fu-systemc:pass
fu-verilog:fail
fve-vhdl:pass
fve-systemverilog:pass
```

Once this job is finished, it triggers a build of other jobs based on the result file of the `Toolchain-generator-codasip_urisc`. The job, which is responsible for that, is called `Sorter`. The role of this job is to process the result file from the generator job and trigger the corresponding downstream jobs. This is pictured in Fig. 6.1.

The trigger of the job is connected to the checkout of repositories and the download of the saved artifacts from the previous jobs. The checkout and download of the artifacts can mean hundreds of megabytes. The jobs that are triggered as downstream jobs perform the functional verification. I trigger three jobs that perform the verification for the Verilog, VHDL and the SystemC language.

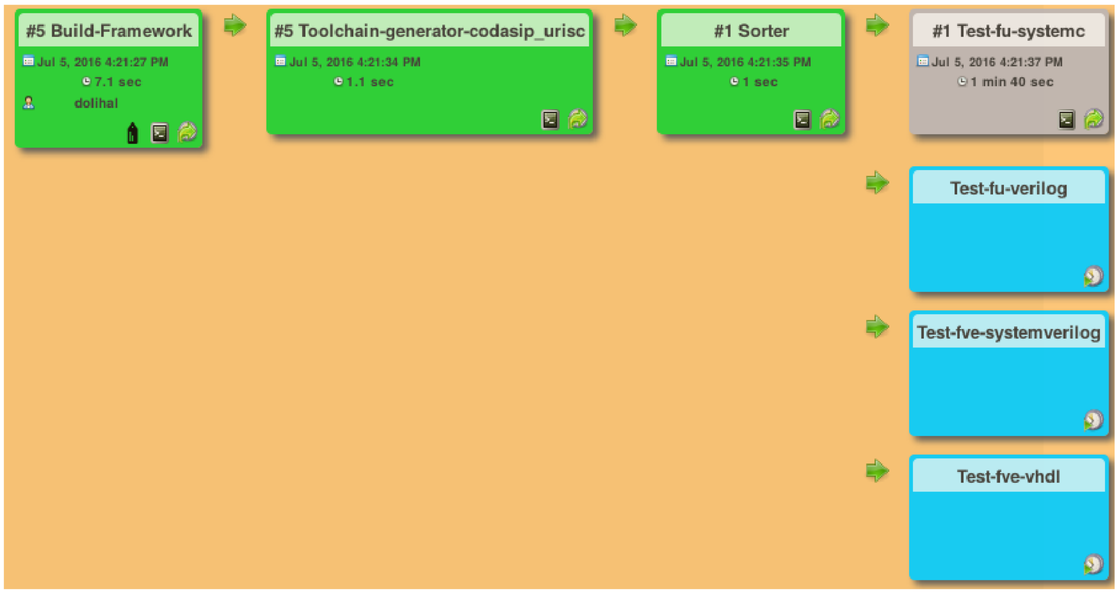

Figure 6.1. Build pipeline with tools generator

I will present the results of the testing which was performed within the Jenkins environment. The results were gained from the Jenkins server in version 1.652.

I have made several experiments with the utilization of the tool generator results and without it. I have also tried various combinations of the successful and unsuccessful jobs. I will present them in several tables and graphs below.

The results in the following Table 6.1 compare the time that was needed for tests of the functional verification with and without the use of the tools generation results.

Use toolsgen results	*Number of fails*	*Time*
YES	0	159m
NO	0	165m
YES	1	106m
NO	1	113m
YES	2	53m
NO	2	62m
YES	3	3m
NO	3	12m

Table 6.1. Comparison of the testing times.

The times in Table 6.1 do not include the time needed for the `Build-Framework` job. It is just the time needed for the testing. From the times we can see that the acceleration is apparent in all cases. The speed-up is gained by the fact that in the case of unsuccessful generation of the environment, I do not have to download files from the git repository and also I do not have to copy large artefacts. The times, when the results of the generator tests were used, do include the time needed for performing the generator tests.

In the case of success, I also significantly reduce the size of the artefacts I have to copy, because I use pre-generated artefacts from the tests of generators. If I do not deploy the tools generator before the main tests, I have to generate a verification environment every time.

6.3 Main contribution

The main contributions of the chosen approach are the following:

- *speed up of testing* - In the case of multiple failed jobs, the chosen approach can save a significant amount of time by not triggering the jobs that would fail, but even when the tests do not fail, the acceleration of tests is apparent.
- *traffic savings* - In the case of a failed job, the approach saves traffic as it prevents the checkout of repositories and the download of artefacts.
- *faster deployment* - In case I use the build automation described by [30], I will be able to deploy and test the new build more often than once a day.

The issues connected to the testing process were described in the article [17]. The use of the results of the generators tests was introduced in the article [14].

Chapter 7

Continuous integration job generator

In this chapter, I will address one of the greatest weaknesses of our project. I very often need to create a new set of tests for a new branch of a certain micro controller or create tests for a completely new core. In such situations, the user can create a whole new set of jobs by hand or find a way how to automatise such a task [15]. I have sketched the possibilities, which are provided by the plugins in the CI server Jenkins and also other solutions in the section State of art.

To create a generator of Jenkins jobs, I need to have good knowledge of the Jenkins job format. The format of the job is in detail described in the full thesis.

7.1 Job generation

The main task that I need to deal with is the generation of the various jobs, which will ensure complex testing of the core. Mainly, I will generate the jobs which test the automatically generated tools. As I plan to control the whole system also from the command line, I wanted to avoid the graphical interface, at least in the first version of the project. I may add the graphical interface in the later versions, but I definitely need to keep the command line interface for the solution to be fully scriptable. This is also one of the reasons, why I cannot use the plugins provided by Jenkins. They have very poor documentation and are primary focused on usage via the web interface.

The whole system consists of three main parts. The first part of the system is the *sniffer*. In my case it works over the *git repository*. Once the generation is triggered, the *job generator* uses *templates* to generate corresponding jobs. I will now give a more detailed description of the aforementioned parts.

7.1.1 Sniffer

I have decided to call this part of the generation process the *Sniffer* as it sniffs in the git repository for new branches. The main role of the Sniffer is to detect the creation of a new branch in the given git repository and trigger the generation.

Although currently the role of the Sniffer is to notify that a new branch has been created and deliver this information to the job generator. The Sniffer has no further intelligence and the whole system is designed in such a way that all decisions should be made in the generator itself. In the latest version, the Sniffer has a shape of the Unix script, which is executed repeatedly by the operation system.

7.1.2 Templates

The second input into the job generator are the templates. I have various kinds of templates as I need to test various parts of the newly developed core. The main areas which have to be covered by test job generation are:

- compiler testing,
- functional verification,
- assembler testing,
- tools generation.

Please note that these are just the areas that need to be covered, not the jobs. Under each domain there is a variety of jobs which are generated and later on executed. There is usually just one template per domain, just in the case of functional verification I need to have several templates, as this area is very vast and I was not able to stick to just one template.

As far as the templates themselves are concerned, they are very simple. The templates are in the XML format, as are the jobs in the Jenkins, and the generated parts are in the form:

```
<string>@NODE_NAME@</string>
```

7.1.3 Job generator

Now when I have described the inputs of the generator, I will move to the generator itself. The job generator consists of several parts that are pictured in Fig. 7.1.

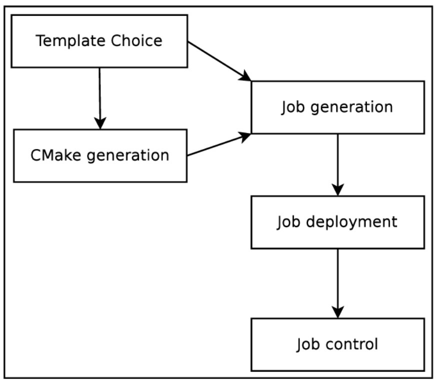

Figure 7.1. Scheme of the generator

One of the first steps is the template selection. This part of the generator works over the configuration file that is present at the specific directory in the model branch which should be tested. I have proposed a simple format of the configuration file that specifies the tested features. The other possibility I have is to automatically detect what features should be tested, but I have chosen the configuration file because some of the features cannot be automatically detected. From the specification file I am able to determine what templates should be used. The specification file has two major tasks:

- to define features that should be tested,
- to specify parameters for the generators.

Once the phase of the templates selection is finished, I need to generate the *CMake* files that will fill the desired information into the templates. *CMake* is a family of tools. These tools are designed for the build, testing and packaging of software. The generated *CMake* files are template specific as each template has different fields. Currently I generate one *CMake* file per template and I perform the generation in the separate directories.

From the two above mentioned inputs I can generate the job. The job generation is in fact just insertion of data into templates. I have decided to do this via the *CMake*, because it is one of the cleanest ways for doing so. The most frequent facts that are generated are the following:

- the branch used for testing,
- the node where the job is executed,
- the bash script and the parameters,
- the job name and the view where the job is placed.

The above mentioned information can be determined in the subsequently described way. The branch is one of the input parameters. It is delivered by the *Sniffer*, but it can also be delivered in a different way, it can be, for example, specified by the user.

The script, which is executed, could be a part of the template, however, this would increase the number of templates significantly. Therefore, I try to determine the name of the script. The name of the script can be determined from the information, which is given in the configuration file.

The job name and view where the job should be placed are also determined from the configuration file and repository name.

The most complicated task is the selection of the correct node where the job should be executed. The management of the nodes is quite a complicated task and is described, for example, here [39].

I have special groups of nodes, for example, for the execution of the verification jobs. The verification jobs require a preconfigured environment, which is present only on certain nodes. For such jobs, I have special templates with the predefined sets of nodes. Nevertheless, for the majority of jobs I do not have to solve such issues. I keep a simple table of nodes which is divided into sections which define what nodes are used for the specific jobs. I choose the jobs with the smallest number of assigned jobs and optionally I modify the assignment manually.

Very often I generate the parameters of the given job into the templates. They are stored in the *parameters* section and later these parameters are used in the *builders* section. However, there are also parameters that are node dependent. The node dependent parameters are defined in the Jenkins environment.

Frequently the generated job needs to use the artefacts from the other jobs. Nevertheless, I try to keep the generator as lightweight as possible and do not want to modify other jobs. The compatibility in this case is assured by the wild cards, and the name of the new job must fit into the wild card.

Once I have generated the jobs, which are needed for the testing of the newly developed branch, I have to upload these jobs to the CI server. For this purpose I use the Jenkins command line interface that performs the job upload and also registers the job.

7.2 Experimental results and contribution

With the current implementation of the simple job generator I have performed a number of tests. I have chosen two typical scenarios. The first case is the generation of a new testing set for the instruction accurate description of a new core. With the IA description corresponds the basic set consisting of tests which test the compiler and the assembler. When the complete description of the new core (instruction accurate, as well as cycle accurate) is created, the full set of tests is generated. The full set adds also tests for functional verification.

The templates, which are needed for the generation of such tests, were added into the template pool. The basic set consists of 3 jobs and the full set consists of 12 jobs.

Method	*Basic set*	*Full set*
Lissom Generator	0,99s	4,2s
Jenkins job generator plugin	2,1s	8,5s
Jenkins DSL plugin	1,3s	5,2s
Manual creation	354s	1417s

Table 7.1. Comparison of creation times.

The Table 7.1 summarizes the comparison of my generator with the common Jenkins generators. I have also added the times needed for the manual creation.

The comparison with the most widely used generators provided by the Jenkins server was made at the following configuration. I used the Jenkins server in version 1.656. The Jenkins server was running on a server with 4 cores Intel i5 and has 8 GB of memory.

It is clear that the Lissom generator is faster than the job generator plugin and the DSL plugin in both tested cases. However, in the case of generation of just three jobs, the times are comparable.

In the case of generation of the big set, the Lissom generator has a clear advantage. It is 1s faster in comparison to the DSL plugin and 4.3 seconds faster in comparison to the job generator plugin. The manual creation of the jobs was slowest in both cases.

Among the main contribution there can be placed:

- *significant speed up of the job generation* - As is clear from the results, the generation of the jobs is faster in comparison to any other generator.
- *higher level of automation* - With the correct configuration the job generation can be provided completely without user interference.
- *wide range of use* - The job generator is dependent only on the xml format of the job, it can virtually generate any type of testing job.
- *no dependency on scripting language* - There is no need to deploy any scripting language, such as Groovy, the jobs are generated from the configuration file.

The topic of the continuous integration environment and the automatic generation of the jobs for such environment was described in the article [15].

Chapter 8

Conclusion

In this thesis, I have addressed the testing of an automatically generated compiler. I have focused on four areas and introduced solutions that help to optimize and automatize the process of testing.

The first area is support of the standard C language library and the process of porting. Due to a good choice of the library, I was able to significantly increase the number of tests that can be used for porting. The raised number of tests gives the developer of the micro controller better possibilities for tuning the compiler and the whole system. I have introduced the universal mechanism that can be used for porting to any platform if the platform is suitable for the C library.

I have also worked on the process of porting with the aim to make it more automatic. I have introduced several ways that make the process of porting more automatic. The number of files that have to be manually changed has been significantly decreased and the whole process of porting is now faster and requires less knowledge.

The second area I have investigated is focused on the test selection mechanism. As was demonstrated, there is currently no mechanism that would suit my needs for the efficient selection of the test cases. I have designed a system of special files that are used for the selection of tests. The scheme is lightweight and robust at the same time. It can be used for any kind of tests and is not platform dependent, so it can be used for any core.

Moreover, I have created a generator of test selection files, which can be used for the generation of new files. The generator can be used once a new core, or just a new version of the existing core, is under development. The generator uses as an input the information contained in the model and the tests themselves that are compiled to the object form. The generation is fast and the accuracy of the results is good.

The area number three is connected with the acceleration of tests which are executed by the continuous integration server Jenkins. I have looked for a way how to decrease the time and space requirements of the functional verification testing and other tests. I have utilised the new kind of tests in our project, the tests of generators. The generator tests are executed as first, and all other tests use the results of the generator tests and, therefore, save time via the pre-generation of the binaries if the tests are successful. If the generator tests fail, the downstream jobs performing the verification tests are not triggered at all and hence save time and space that would otherwise be spent on the checkout of files.

Last but not least, I have sketched a simple generator of the Jenkins jobs that would suite our needs in the Lissom project. I need a generator that can be started by various ways, which is lightweight and can generate all kinds of jobs. This was one of the basic requirements, which was not met by any plugin that is currently available for the Jenkins.

I also wanted the tool to be at least partly independent of the Jenkins as it is not rare that the plugins do not cooperate well.

The current implementation of the generator is dependent just on the internal representation of the job. This is not a problem, as it is very simple to deploy new templates. At the same time, the internal job representation is not likely to change as it would imply changes in all plugins currently used by the Jenkins.

I put the generator under tests and the gathered results are very positive. As far as the speed of the generator is concerned, it cannot be matched by any tool that is currently available.

The implementation of the generators and other tools was performed in the Python language, so the solutions are easily extensible.

8.1 Future work

In the future, I would like to apply the use of the tool generator results also on other kinds of testing, such as the compiler or the assembler. I believe that I could gain some time savings in the case of application. Via this approach it should be possible to achieve speed for every group of tests that is more complex.

The implementation of generator of the testing jobs could also be extended. I could add support for the copy artefacts section and also support for the folders plugin that we currently use in our project. I would also like to find ways how to improve the speed of the generation.

It would also make sense to introduce a code generator into the testing process. It could uncover interesting new bugs in the automatically generated compiler.

Bibliography

[1] Job DSL Plugin.
<https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin> (July 2016), 2016.

[2] Job Generator Plugin.
<https://wiki.jenkins-ci.org/display/JENKINS/Job+Generator+Plugin> (July 2016), 2016.

[3] ANSI: INCITS/ISO/IEC 9899-1999 (R2005). <http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS/ISO/IEC%209899-1999%20%28R2005%29/> (April 2016), 2016.

[4] ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, Issue C. 2014.

[5] Arquilian: Arquilian. <http://arquillian.org/> (March 2016), 2016.

[6] Autotest: Autotest. <http://autotest.github.io/> (March 2016), 2016.

[7] Bennett, J.: Howto: Porting newlib, A Simple Guide. 2010.

[8] Charvat, L.; Smrcka, A.; Vojnar, T.: Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description. In *Proceedings of the 13th International Workshop on Microprocessor Test and Verification (MTV 2012)*, Institute of Electrical and Electronics Engineers, 2012, ISBN 978-1-4673-4441-8, pp. 6–12.
URL <http://www.fit.vutbr.cz/research/view_pub.php?id=10135>

[9] Cucumber: Cucumber. <https://cucumber.io//> (March 2016), 2016.

[10] De Micheli, G.; Rolf, W., E.and Wolf: *Readings in Hardware/Software Co-design.* Morgan Kaufmann, 2001, ISBN: 9781558607026.

[11] Dolihal, L.; et al.: Use of Architecture Description Language ISAC fo ASIP Design. In *In Proceedings of Eighth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*, European Network on High Performance and Embedded Architecture and Compilation, 2012, ISBN 978-90-382-1987-5.

[12] Dolihal, L.; Hruska, T.: Porting of C library, Testing of generated compiler. In *In Proceedings of The Sixth International Multi-Conference on Computing in the Global Information Technology*, International Academy, Research, and Industry Association, 2011, ISBN 978-1-61208-008-6, pp. 125–130.

[13] Dolihal, L.; Hruska, T.: Semiautomatic Porting of the C Library. In *In Proceedings of International Conference on Computer Science, Computer Engineering, and Education Technologies*, International Academy, Research, and Industry Association, 2014, ISBN 978-1-941968-02-4, pp. 86–89.

[14] Dolihal, L.; Hruska, T.: Overview of the testing environment for the embedded systems. In *In Proceedings of The third International Conference on Green Computing, Technology and Innovation*, International Academy, Research, and Industry Association, 2015, ISBN 978-1-941968-15-4, pp. 86–89.

[15] Dolihal, L.; Hruska, T.: Automatic Job Generation for Compiler Testing, Testing of Generated Compiler. In *In Proceedings of The Eighth International Conference on Advances in System Testing and Validation Lifecycle*, International Academy, Research, and Industry Association, 2016, ISBN 978-1-61208-500-5, pp. 1–6.

[16] Dolihal, L.; Hruska, T.; Masarik, K.: Testing of an automatically generated compiler, Review of retargetable testing system. In *International Journal on Advances in Software, 2012*, year 2012, International Academy, Research, and Industry Association, 2012, ISSN 1942-2628, pp. 15–26.

[17] Dolihal, L.; Hruska, T.; Masarik, K.: Testing System for the HW/SW Codesign Toolchain. In *In Proceedings of Eighth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, NOVPRESS, 2012, ISBN 978-80-87342-15-2.

[18] Dolihal, L.; Hruska, T.; Masarik, K.: Usage of simulators in testing system. In *In Proceedings of Industrial Simulation Conference 2012*, EUROSIS, 2012, ISBN 978-90-77381-71-7.

[19] Gatliff, B.: Porting and Using Newlib in Embedded Systems. `<http://neptune.billgatliff.com/newlib.html>` (March 2016), 2016.

[20] GCC: GCC Compiler website. `<https://gcc.gnu.org/>` (Fedruary 2016), 2016.

[21] Git: git. `<https://git-scm.com/>` (February 2016), 2016.

[22] Hat, R.: Red Hat. `<https://www.redhat.com/>` (August 2016), 2016.

[23] Hubertz, J.: *Softwaretests mit Python.* Springer, 2016, ISBN 978-3662486023.

[24] Husar, A.: *Programming of reconfigurable systems using a higher programming language.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2014.

[25] Jenkins: Jenkins. `<https://wiki.jenkins-ci.org/display/JENKINS/Home>` (March 2016), 2016.

[26] Kroustek, J.: *Retargetable analysis of machine code.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2014.

[27] Lier, F.; Wienke, J.; Wrede, S.: Jenkins for FloBI–A Use Case: Jenkins & Robotics. In *Jenkins User Conference*, 2013.

[28] Lissom: Project Lissom Webpages.
`<http://www.fit.vutbr.cz/research/groups/lissom/>` (August 2014), 2014.

[29] LLVM: LLVM Compiler website. `<http://llvm.org/>` (Fedruary 2016), 2016.

[30] Lukasova, M.: *Build Paralelization in Jenkins Environment.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 201.

[31] Masarik, K.: *System for hardware-software codesign.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2008.

[32] MiBench: MiBench. `<https://github.com/embecosm/mibench>` (June 2016), 2016.

[33] Newlib: Newlib. `<https://sourceware.org/newlib/>` (March 2016), 2016.

[34] Perennial: Perennial C Compiler Valication Suite.
`<http://www.peren.com/pages/products_set.htm>` (August 2014), 2015.

[35] Prikryl, Z.; Kroustek, J.; Hruska, T.; aj.: Fast Just-In-Time Translated Simulation for ASIP Design. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2011, ISBN 978-1-4244-9753-9, pp. 279–282.
URL `<http://www.fit.vutbr.cz/research/view_pub.php?id=9567>`

[36] pytest: pytest fixtures: explicit, modular, scalable.
`<http://pytest.org/latest/fixture.html>` (July 2016), 2016.

[37] Rowen, Chris and Hennessy, John , and Christensen, Clayton M. and Leibson, Steve: *Engineering the complex SOC : fast, flexible design with configurable processors.* Prentice Hall Modern Semiconductor Design Series, Upper Saddle River: Prentice Hall, 2004, ISBN 0-13-145537-0.
URL `<http://opac.inria.fr/record=b1108184>`

[38] Selenium: Selenium. `<http://www.seleniumhq.org/>` (March 2016), 2016.

[39] Skala, M.: *Virtual Machine Management System.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 201.

[40] Synopsys: Processor Designer. `<http://www.synopsys.com/systems/blockdesign/processordev/pages/default.aspx>` (August 2014), 2014.

[41] Teich, J.: Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 2012.

[42] Wang, L.-T.; Chang, Y.-W.; Cheng, K.-T. T.: *Electronic design automation: synthesis, verification, and test.* Morgan Kaufmann, 2009.