Overview of the testing environment for the embedded systems

Ludek Dolihal and Tomas Hruska
Faculty of Information Technology, Brno University of Technology
Bozetechova 1/2,
612 66 Brno, Czech Republic
{idolihal, hruska}@fit.vutbr.cz

ABSTRACT

For testing of automatically generated C compiler for embedded systems on simulator, it is useful to have a C library support. Testing programs written in C very often use I/O operations and other functions provided by the C library. Hence not having the library, the number of programs that can be executed is very limited. But the C library can be used for the wide variety of tests not just for the C compiler. We will try to point out the role of the library in the whole process of testing and give an overview of the testing system that is used for testing of the complex embedded systems.

KEYWORDS

Embedded systems testing, C library, compiler testing, simulation, hardware/software co-design, integration, Newlib, Lissom, Codasip.

1 INTRODUCTION

This article is going to discuss the area of hardware software codesign. Hardware software codesign deals with the design of the new embedded systems. This kind of systems can be found in wide variety of devices such as network routers or printers.

The embedded systems consists of one or more application specific processors (ASIPs). Each processor usually takes care of single specific task and therefore is highly optimized for this task. The optimization is also the main difference from general purpose processors (such as x86 family) that has to take care of very wide range tasks.

The production of ASIPs in 2012 created over 98% of overall processor production. Hence this area is extremely important. The technology used for creation of any ASIP is called System on the Chip (SoC). Such technology allows the integration of several ASIPs on one chip together with peripherals such as memories, buses, etc.

The development of today ASIPs must be done in a very short period of time. To do so, it is common to use the tools for hardware software codesign. In the core of such tools is typically some hardware description language (HDL). The development is done in a modern integrated development environment (IDE) that allows the designer to generate all the necessary tools, such as compiler, assembler or simulator.

As the main aim of the Lissom project [1] (now also Codasip project- www.codasip.com) is hardware software co-design, we need to ensure that the tool for the HW/SW has the full functionality and contains a minimum of errors. The main priority of the testing system is to discover the errors as soon as possible and shorten the time for fixing it.

This paper is going to give an overview of the testing system that is currently used in our project. It is organized in the following way. The second section gives a brief overview of the Lissom project, section three discusses the related work, section four deals with tools oriented testing, section five with model oriented testing and section six concludes the paper.

2 OVERVIEW OF THE PROJECT

The Lissom project focuses mainly on the hardware software co-design. As we try to provide the best possible solution, we have
created also the C compiler, because the C is still the main development language for the embedded systems. Besides the C compiler there are various other tools generated from the processors description. The processor is described in the ISAC language [2]. Amongst the generated tools are:

- simulators,
- assembler,
- disassembler,
- debugger and
- verification environment.

The generated simulator was thoroughly described in the article [2] and verification environment description can be found in [3].

From the description in the HDL we are also able to generate the hardware description. Currently we support the VHDL language as well as Verilog and SystemC.

What makes our tool quite extraordinary is the ability to deliver the compiler with libraries. Currently we are able to deliver the C compiler for various platforms that match several basic requirements, as it is not possible to create compiler for every architecture. For the generation of the compiler is used the description in the form of HDL. This approach is standard for all platform dependent tools.

As is apparent, the tools are usually after several steps generated from the specification in the ISAC language. This description is crucial for all parts of the design. There is a necessity to test the generators of the tools thoroughly.

It should be mentioned that together with the tools goes also hand in hand an integrated development environment (IDE). The IDE is based on Eclipse project and is customized to considerable measure.

3 RELATED WORK

There are no papers published on the whole testing system. At least not from the overview perspective. Hence it was decided for this section to take a bit deeper look into the problematics and focus on one special part of the testing system. We have chosen the porting of the C library that plays crucial part in the testing of the whole system. The library used in our case is called the Newlib.

The basic approach to the porting of the Newlib library is described at official pages of the project [4]. The pages give a quite detailed guide to the process of the creation of the port for the new architecture. When we went through the library we found several ports made by third parties. However, the description contained on the manual that can be downloaded from the project pages is far from being automatic. The user has to conduct quite a lot of manual steps to get the functional version of the library.

The automatic process of porting is in the core of the paper Automatic Porting of Binary File Descriptor Library [5]. This approach is quite similar to the process that is going to be described in this paper. Nevertheless, the mentioned paper describes port of the GNU Binutils core. The BFD library is used for the manipulation of the object files, so its purpose is completely different of the Newlib library. Unfortunately the paper gives just an overview of the process, it is not described in detail in the paper.

The Unisim [6] project should also be mentioned. It was developed as an open simulation environment which should deal with several crucial problems of today simulators. It solves the lack of the interoperability problem by wrapping them into the modules. Though this may seem to be a little out of our concern the idea of the interface within the simulator that allows addition of any library is quite interesting. We use modules in a completely different way in our approach. In the first case we use them inside the library itself. The Newlib library has a modular composition. It enables the user to switch on or off certain parts of the library.
4 TOOLS ORIENTED TESTING

There are two main points of view when we talk about the testing. First of them is the tools-oriented testing.

In the tools oriented testing, we need to ensure, that the generated tools as well as the generators itself 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.

Lets have a look at the generators first. The generators are in our case triggered via a command line interface. We have created in Python a set of classes that enables us to perform full test of the command line functionality. This testsuite in combination with various models gives us very strong tool for ensuring that our generators are stable. The testsuite is highly modifiable. We can also very easily enhance this testsuite with performance tests and stress tests. The testsuite can be executed in the mode, that tests all combinations of parameters that are legal. The results of the generators testing is one of the inputs into the testing of the generated tools. This attitude is sketched on the Figure 1.

When we get to the testing of the specific generated tool we first have a look at the generators tests. In case we find out any problems during the generation, we either skip the tests as a whole or we need to be more careful.

The great part of the generated tools is designed by the software development kit (SDK). The SDK contains the GCC like compiler driver. In our case, it is the solution based on LLVM [7]. The LLVM compiler driver is called Clang. In the SDK, there are two main parts that are automatically generated, hence they are error prone. These two parts are assembler and compiler backend.

Now comes the time for the results of the generators testing. Because we have the results from the various platforms we can schedule and perform the test of given tool, in this case the assembler only on the working platforms. In case there have been issues with generation on all platforms we skip the whole process of testing. In case the tool has been generated correctly, assembler is put under tests. It is thoroughly tested by the randomgen. The randomgen program generates automatically from the processor description valid programs, that the assembler binary must be able to process. The randomgen application is also automatically generated so the paragraph above also applies for it. But this is only one way the assembler is tested. This way we ensure that the valid constructions will be assembled without problems. Nevertheless, the assembler is also tested within the compiler driver.

![Figure 1. Scheme of the system](image)

Now we get to the testing of the compiler backend. The input of the backend are the files that are in the certain kind of the internal representation of the compiler driver and the output is the assembler. Here we can see the very close interconnection with the assembler, which is responsible for transformation of the assembly language to the object files.

We have several ways of testing of the compiler backend. The first line consists of simple tests, taken from various testsuites such as GCC torture testsuite. These simple tests are meant for fast debugging of the backend.
There we can also utilize the results of the generators testing. Not only that we have to check that the backend together with the compiler driver was generated but we can also check if the necessary libraries were compiled. If not, we can choose only a subset of tests and shorten the testing time. In case we do not have the Newlib library compiled we can save up to several hours of testing.

Second line consists of benchmarks. The purpose of this test is to tune the performance of the compiler. They can also be used for debugging, but it is not as comfortable as in the previous case. What is important in this case is the fact, that we very closely observe a number of cycles that are needed for each benchmark. In case that we have 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 an ultimate tests that the compiler as well as the model contain minimum of errors.

The results of the generators testing come to use in this case as well. In addition to all the tools that are required for the tests of simple programs we also require a presence of the Newlib library.

For execution of all three categories of the programs is used simulator.

5 MODEL ORIENTED TESTING

Another point of view at the testing system is from the angle of models. The model developer expects that the tools work. He is interested in his processor design, and needs to get the results of the testing all in one place.

The needs of the model designers are partly met via the different results scheme. We support the scheme that is tool oriented. But for the designers is more suitable the model oriented scheme that is currently under development.

The errors in the model are very often revealed in the phase of the tools generation. The tools contain various checks that ensure, that the tool can be generated. For example the compiler backend can not be generated in case the model does not contain certain instructions (for example jump and so on).

Generally we can say that for the models oriented point of view the generators testing is very important.

The most model oriented tests that we currently deploy is the functional verification. The functional verification function is to verify the equivalence of instruction accurate (IA) and cycle accurate (CA) model. The IA model describes the controller on the level of instructions, while the CA model is more precise. It describes the set of operations that represents the separate actions between the clock cycles. From each description is generated a tool. In case of IA we generate simulator and in case of CA we use a generated verification environment. We execute the same program on both and then we compare the results. Such tests are performed when the model are stable as it uses the tools from IA and CA description. This tests help us to discover the differences in model descriptions.

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

But here we can also utilize the knowledge we have from the generators testing. Moreover we need the results from the compiler testing as we use the compiled binaries for execution.

Recently we also added into our system a synthesis tests. The output of this tests gives us basic information about the area that the core will occupy. We perform such test with weekly frequency. The output of such tools together with the benchmark tests of the compiler are the base for power consumption estimations.
6 CONCLUSION

In this short paper, we tried to give an overview of the modern testing system for embedded systems. We tried to sketch the problematics of testing of the tools as well as testing of the cores itself. These perspectives are very different. The first one is focused mainly on the SDK and its two main generated parts the assembler and compiler backend. The testing of generators stands a bit aside. This tools are not critical from the point of view of performance. On the other hand we must watch closely the cycles of the compiler backend as it might affect the power consumption.

The testing of the cores is a completely different issue. There is a constant pressure on low consumption which we can estimate thanks to compiler benchmark tests and area that we get from the synthesis tests.

However both of this attitudes take advantage of the generators testing.

ACKNOWLEDGEMENTS

This work was supported by the IT4 Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

REFERENCES


