MIND THE GAP
TEACHING COMPUTER AND COMPILER DESIGN IN LOCKSTEP

Edward Alexandru Todirica and Christian W Probst

DTU Compute, Technical University of Denmark

ABSTRACT

Learning often only occurs when we are confronted with two, possibly contradicting scenarios that should be explainable by the same approach. In this article we present a design-build project that exploits this situation for teaching computer science students about processors, compilers, and their interaction.

The design-build project is based on two parallel courses in the third term of a computer systems engineering study line: one course considers both the hardware and software for a computer system, and the other course focuses on the compiler used by that computer system. In the hardware/software course, the students learn how a computer works from the level of transistors all the way up to the level of high level programming languages. They implement a computer from the ground up on an FPGA and then write programs for it in machine code, assembly, and C. During this course they also learn about some of the internals of the C compiler for the processor used in this computer system. In the compiler course the students implement a compiler for a high-level language, a subset of Java, that generates native code for the processor they develop in the hardware/software course.

The students thus explore the complex task of generating executable native code from two directions: in the hardware/software course bottom-up, and in the compiler course top-down. This approach enables deep insights into and better understanding of the processor, the compiler, and the execution of the generated code.

In an intensive CDIO project period at the end of the hardware/software course, the students get the loosely defined task to design and build a small computer-based system using the processor introduced in the hardware/software course, and to program an application for it. Although the project is closely linked to these two courses, many students include topics from other courses, such as knowledge from a course on objected-oriented analysis and design for interaction with a host system, knowledge on databases, or from statistics.

KEYWORDS

Computer Science, project work, peer evaluation, Standards 5, 7, 8

INTRODUCTION

In this article we discuss a design-build project based on two courses, a hardware/software and a compiler course in the third term of a computer systems engineering study line. In the hardware/software course, the students learn how to build a processor from the ground up, by implementing it on an FPGA, and learning how a C compiler for this processor works internally. An FPGA is a chip that allows quick prototyping of complex digital circuits, e.g., memory, CPUs, or graphics cards. In the compiler course, they learn how to implement a compiler for a high-level language, a subset of Java that generates native code for the processor they develop in the hardware/software course. The students thus explore the rather complex topic of generating executable native code from two directions: in the hardware/software course bottom-up, and in the compiler course top-down. Combined, these two courses enable deep insights into and better understanding of the processor, the task of compiling, and the languages C and Java.

The overall aim of the courses and the associated project is to teach the fundamentals of how a processor and an embedded computer system work, both from hardware and from a software point of view. They run in parallel for the term, followed by an intensive, 3 week project period, where the students get the loosely defined task to design and build a small computer-based system using the processor developed in the hardware/software course, and to program an application for it.

Although the project is closely linked to these two courses, many students include topics from other courses in their project. Some groups use knowledge from a course on objected-oriented analysis and design for interaction with a host system; others use knowledge on databases, or from statistics. In an ideal world they would add project-specific support to the Java compiler developed in the compiler course. However, since the project is so loosely defined, this component is hard to control, and in our view should only be an option.

During the course of the third term we experience a clear first drop of excitement due to the complexity of the topic, which then is overcome by excitement due to the possibilities, and the increased understanding due to the two complementary approaches to explaining both the low-level execution and the high-level languages. Both courses have been re-designed in recent years to accommodate the complementarity.

In this article we present the course construction and discuss dependencies between the two courses, and discuss the general principle of combining complementary courses that cover similar material.

The rest of this article is structured as follows. In the next section we briefly describe the general curriculum of the study line, the two courses belong to, and the learning objectives of the courses. We then discuss the interaction between the two courses as well as the differences, followed by a presentation of the results of the individual courses. Before concluding the paper with a summary and final remarks, we present the course results and discuss student evaluations of the courses.

COURSE DESCRIPTION

Before discussing the two courses, we briefly present the study line that the two courses belong to.

Study Line "Diplom IT"

The two courses we consider in this article are part of a B.Eng. study line on both hardware and software and the interaction between them (computer systems engineering). The study line thus covers software engineering and core computer science competencies as well as electronics and embedded systems, where one or more computers are embedded in a device and work together, often hidden from the user. The development of this kind of systems requires systematic approaches for requirements engineering, design of system models, system implementation, and finally the deployment of the system, thus clearly mapping to the core competencies in the CDIO syllabus (E. Crawley, 2007).

Term	5 ECTS	5 ECTS	5 ECTS	5 ECTS	5 ECTS	5 ECTS
1	Mathematics		Electronics	Development methods for IT systems	Introductory Programming	
2	Discrete Mathematics & Databases	Digital Systems	Data Communi-cation	Algorithms and Data Structures	Advanced Programming	
3	Probability and Statistics	OOAD and Databases		Compilers	HW/SW Programming	
4	Parallel and Realtime Systems		Model-based Software Develop-ment	Distributed Systems	CDIO Project	
5	Electives					
6	Internship					
7	Electives		Exam Project			

Figure 1. Study plan of the study line Diplom IT.

Figure 1 shows the study plan for the respective study line (Jens Sparsø, 2011). Throughout the first 4 obligatory terms, students get a basic education consisting of elements from both software engineering and electronics and embedded systems, eventually leading up to the CDIO project in the fourth term. In each term several courses contribute to a (smaller) CDIO project; in the third term these are the courses on Compilers and HW/SW Programming, which are covered in this paper.

The Bottom-Up Course – HW/SW Programming

The overall aim of the course HW/SW Programming and the associated CDIO project is to teach the fundamentals of how a processor and an embedded computer system work, both from a hardware and from a software point of view. The course is divided into two phases: during the semester the course covers micro-architecture and hardware implementation of a simple processor (4 weeks), low level programming using machine code and assembly (4 weeks), and high-level programming in C (4 weeks). During the subsequent intensive 3-week period students design and implement a small, embedded computer system and implement a small application using it – typically some form of graphics-based game.

Figure 2 shows the block diagram of the hardware platform that the project is based on, which the students implement on an FPGA (Chu, 2008). The core LC-3 based computer system (Patel, 2004), along with the necessary software tools (assembler, simulator etc.), are introduced during the semester. LC-3 stands for “Little Computer version 3“ and is the name of a very simple CPU, containing less than 20 instructions. A few of the instructions used by this CPU were added at DTU in order to improve the size of the generated code. For the project, students are provided with VHDL-code for the LC-3 processor (developed for this course) and VHDL code for the other hardware blocks are drawn from (Chu, 2008). VHDL is the hardware description language used to create digital electronic circuits inside and FPGA. The task of the students is to integrate the components needed, reflecting a typical component-based approach to hardware design. The block called “other hardware” typically comprises a simple VGA display driver and controllers for different types of input/output devices (keyboard, mouse, serial connection, etc.).

As mentioned before, the project specification is very open: “Design and build a small

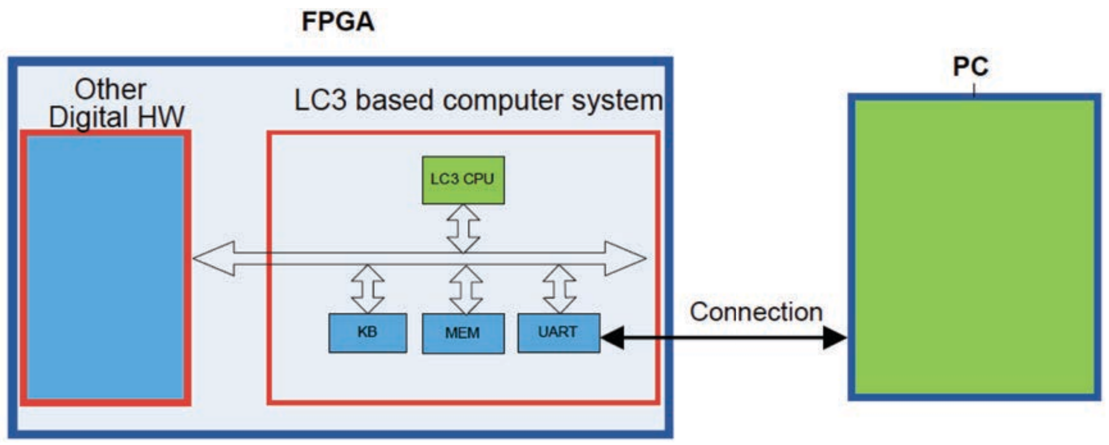

Figure 2. General structure of the LC3-based computer system used in the hardware/software course.

computer-based system using the LC-3 processor and program an application for it". Typical examples are graphics-based arcade-style games like packman, pong, tetris, hangman, space invaders, black jack, breakout etc. (Todirica, 2014), but many groups also do more innovative projects, for example controlling a robot, a fractal renderer, and a piano application.

The learning objectives for this course are

- Develop a general understanding of how a simple embedded computer system is organized and works by considering the components involved and the layers of abstractions used to handle the complexity in such a system.
- Design, implement, debug, test and document a computer system for an embedded application consisting of hardware and software components.
- Describe the different layers of abstraction used in a computer system.
- Construct a simple computer system (containing processor, memory and input/output devices).
- Partition the design of a computer system of some complexity into smaller and manageable parts of hardware and software.
- Explain memory mapped input/output and use it to allow the processor communicate with devices on a computer system.
- Construct small programs that allow the processor to control input/output devices.
- Explain and construct simple programs written in assembly or machine code.
- Construct software programs using a high level programming language.
- Describe and use pointers in programs, for example to create simple dynamic data structures like linked lists.

The Top-Down Course – Compiler Construction

The overall aim in the course "Compiler Construction" and the associated project is to teach the fundamentals of translating high-level languages into native code, and how a compiler realizes this task, with a focus on the software side of this process. During the semester, the course teaches scanning, parsing, and intermediate representations (2 weeks), semantic analysis (4 weeks), processors, virtual machines, and runtime environments (2 weeks), and code generation (3 weeks). The course naturally connects to the aforementioned 3 weeks project of the course HW/SW Programming, where students can use the knowledge obtained in the Compiler Course for their project.

In the project associated with the course, students get a rudimentary, working compiler for a subset for Java (Appel, 2002). The task is to add functionality to the phases of the compiler by adding support for new constructs from scanning to code generation. This training in extending an existing compiler directly supports the 3-week project, as the students have had intensive training in adding functionality, which they can use in the project as well.

The learning objectives for this course are

- Understand the principles of compilers and virtual machines.
- Use and construct software tools to implement a working compiler.
- Explain the different phases in compilation and execution;
- Operate selected tools relating to the compiler phases (e.g., lexers, parsers);
- Explain the different elements in the description of a programming language;
- Derive specifications of the compiler phases, given a textual description of the syntax of a programming language;
- Implement an analysis and code generation phase, given a textual description of the semantics of a programming language; and
- Develop a working compiler.

INTERACTION BETWEEN THE COURSES

When students follow a university course, they sometimes have a hard time understanding the value of the topics that they are learning. This is especially true in the first years of study, when students feel they have relatively limited knowledge about the subject area that their study line is covering. Putting the topics covered into context makes it much easier for them to understand the topics' contribution to their studies, and the topics' relation to each other.

The hardware/software course provides a basic understanding of how a computer and a compiler work. It enables the students to understand the relationship between the two, and appreciate for example the value of knowing in depth how a compiler works. The compiler course, on the other hand, puts focus on mapping high-level concepts to hardware, and thus complements the topics covered in the hardware/software course.

The hardware/software course is a CDIO course that integrates the other courses from the term (see Figure 2). At the end of the term the students implement an open-ended 3-weeks project that offers them the opportunity to conceive, design, implement, and operate. In the intensive 3-weeks period, the students work full time for this project without any other courses running in parallel. They are offered the opportunity to use their own Java compiler that they have developed in the compiler course, if they want to. Even if they prefer to use the C compiler provided with the system, they still benefit from the knowledge that they have gained during the compiler course. Since the LC-3 (Patel, 2004) is a processor used only for teaching, the resources used for developing a C compiler for it were very limited. As a result the compiler has its own limitations and bugs. The knowledge that the students gain in the compiler course helps them understand and overcome these limitations.

The two courses have thus a synergy effect, which allows students to create impressive projects in only 3 weeks. Without the hardware/software course they would not see the need for studying compilers, and without the compiler course they would not be able to overcome the limitations of the C compiler.

At the very end of the 3-weeks project, the students present their implementations to the institute and the general public in an open house event. The visitors are invited to try out the various projects and to vote on the projects that they like best. The winners of the popularity vote are announced at the end of the open house event. The students see themselves like small start-ups where they try to come up with a relevant product idea, create the product and make it usable to the general public, this way covering all the aspects of CDIO in a fun and rewarding way.

COURSE RESULTS AND EVALUATIONS

Both courses have been redesigned from ground up with the introduction of the new CDIO study plan in autumn 2008 (Jens Sparsø, 2011). In this section we briefly describe the resulting changes in course evaluations.

Courses taught at the Technical University of Denmark are evaluated by the students at the end of the course. The students are presented a series of statements related to the course with which they have to agree or disagree. There are 5 levels, from 1 to 5, where 1 represents strongly disagree and 5 represents strongly agree. In order to see the impact of the changes for the two courses, we have collected the averages for two of the statements that we considered most relevant for measuring this change. The first statement is: “I think I am learning a lot in this course” (S1), the second statement is: “In general, I think this is a good course” (S2).

For each statement we show the average values for S1 and S2, the number of students enrolled in the course, the number of responses received in the evaluation, and the response rate.

The Hardware/Software Course

The hardware platform used for this course has always been centred around an embedded processor and an FPGA. The pre-CDIO version of the course used a specific microcontroller, which meant that the course was rather technology oriented, where students would spend a significant amount of time learning how to use that specific microcontroller. The project at the end of the course was predefined, which meant that all the students had to implement the same thing.

The new version of the course teaches students how a computer system works in general, using as a basis for discussion the LC3 processor. This represents a paradigm shift where the focus has changed from *technology* to *concepts* and general understanding of how a computer works. This enables students to work with any type of microcontroller rather than only with a specific one, as it was the case with the old version of the course. Another big motivating factor for the students is that the project at the end of the course is an open-ended CDIO project.

The computer that the students build on the FPGA is similar in complexity with home computers from the 80’s like Commodore 64 and Sinclair Spectrum. Not surprisingly, the applications that students choose to run on the computer they’ve created are similar to the typical applications from that era. Most of the students choose to implement games with simple 2D graphics that are controlled via keyboard, mouse, or even exotic devices like racing wheels or PlayStation buzzers that were modified to connect directly to the FPGA. In some other projects the computer built by students is used to control a robot arm, to draw fractals or even to implement a virtual piano. External resources such as the Internet, databases, or a sound card, are accessed via serial communication to a PC. Figure 3 shows screenshots from the projects done in January 2014; projects from the last few years are available on the course web site (Todirica, 2014).

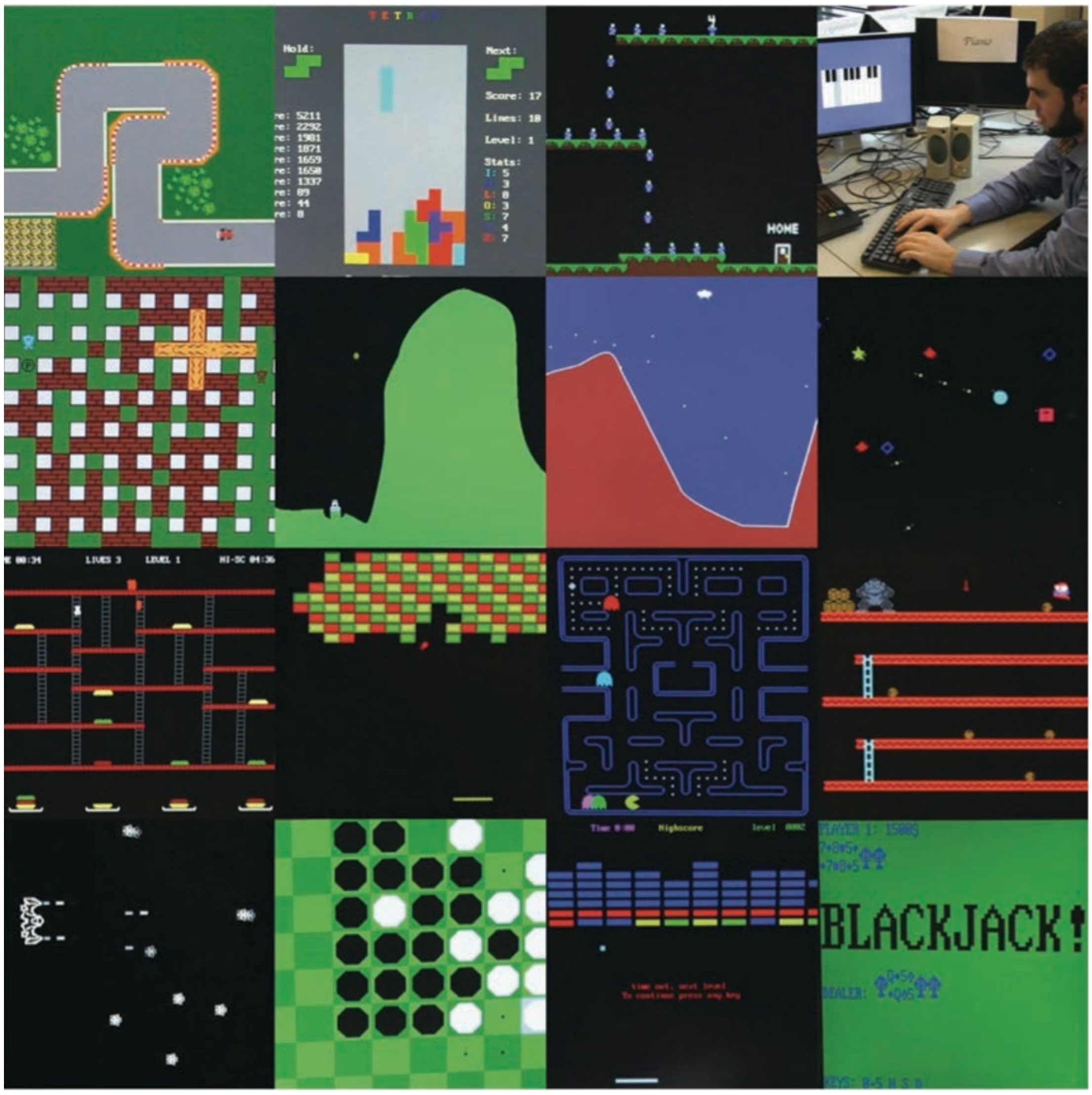
Figure 3. Student projects implemented in January 2014 for the Hardware/Software CDIO project.

Figure 4 shows the evaluation results for various semesters. Here are a few notes that can help explain some of the values from Figure 4:

- The same lecturer has been teaching this course since before 2008 to present day, so the numbers in Figure 4 reflect to a large extent the changes done in the course due to the introduction of the CDIO study plan.
- In spring 2009 the old version of the course (pre-CDIO) was held for the last time, in order to accommodate students who had either failed the course in the past or did not follow the course at the scheduled time in their studies.

Semester	Average S1	Average S2	Number of Students	Number of Responses	Response rate
Spring 2008	3.2	3.1	48	14	29%
Spring 2009	3.5	2.8	29	8	28%
Fall 2009	4.3	4.3	44	18	41%
Fall 2010	4.5	4.3	39	21	54%
Fall 2013	4.1	4.0	63	33	52%

Figure 4. Evaluation for the Hardware/Software course

- The new CDIO version of the course has been running for the first time in fall 2009. There is a clear inflection point between the spring and the fall version of 2009.
- Numbers from Fall 2013 are provided to show the recent status of the course. The dynamics of the course was somewhat different due to the large number of students enrolled in the course (about 50% more than in the past). This could explain the perceived drop in the average values for the two statements considered in Figure 4.

To support the numbers from Figure 4 we cite some comments from students:

- VERY interesting 3 weeks project. Nice that we could choose whatever game we would and still get help.
- This course gave a good understanding of C and assembly and the executing of it on a simple computer.
- My interest in hardware has increased a lot due to this course and my overall understanding of my studies in general have been improved a lot.
- Having emulators/simulators increased the productivity of our group immensely.
- It was a bit difficult to connect all the pieces of knowledge from the 13 week period partly because it was very dry knowledge and partly because it was from so many areas, however the 3-week period connected all the pieces and I felt like I learned a lot from the course. Also the 3-week period has been the hardest and most difficult by far, however I felt like it was worth and I learned a lot.
- The course is filled with a lot of content, which makes it interesting.

The Compiler Course

The compiler course was completely re-designed in 2007, when the current teacher took over the course. The earlier version had been integrated into a bigger course that also covered algorithms and data structures. In the compiler project in the earlier version of the course, students had to find errors in an existing code base, a task that by many students in the evaluations was described as boring and unintuitive. With the migration towards CDIO, the course was changed into a real compiler course; it provides the necessary theory as well

Semester	Average S1	Average S2	Number of Students	Number of Responses	Response rate
Fall 2009	2.5	2.0	63	24	38%
Fall 2010	3.8	3.3	48	15	31%
Fall 2011	3.6	3.3	46	10	22%
Fall 2012	4.0	3.6	46	7	15%
Fall 2013	3.8	4.0	71	15	21%

Figure 5. Evaluation results for the Compiler course.

as hands-on experience with compilers to enable students to develop and maintain compilers in industry. This results in a more practical version of the course, with relevant project work. While many students still are overwhelmed by the project work, as can be seen in the evaluation shown in Figure 5, the overall comments are positive:

- The projects give good insights into the functionality of a compiler.
- This is an exciting topic!
- The projects that built on each other make it easy to understand compilers.
- I understand now what a compiler does and why.
- The contrast with the HW/SW programming course makes it even clearer how programs are executed.

COMPLEMENTING COURSES AS SOURCE OF LEARNING

Behind the success of combining the two courses lies an insight that we gained as a by-product. The two courses described above contribute to a design-build project. In this project, students approach the same advanced topic – the execution of native code on a processor – from two different directions. In the one course, they come from below, and learn the functionalities provided by the computer system on which the code is executed. In the other course, they come from above, and learn the functionalities of the compiler, that is generating the code that is executed on the processor.

The two, differing scenarios of code execution provide the students with different explanations, how code execution works. Both scenarios achieve the same goal – executing code – but they do so in different ways. Some of these differences are quite obvious, for example the difference in the high-level languages, other are very subtle, for example the organization of activation records and their layout in memory.

In this setting, towards the end of the 13 weeks period, the students face a situation that forces them to consolidate two different explanations. It is here that deep learning occurs and students have a revealing experience of understanding each individual topic covered in the courses as well as the overlapping area much better.

An interesting question is, whether this effect can be replicated throughout the term, or at least at several points through the term. In the setting described in this paper this is hindered by the fact that the two courses come from two different directions. This means that they only in the very end cover the same topic at the same time. For other topics, this effect can only be simulated: the course that covers the topic later needs to repeat the material from the course that has covered the topic before. The required substantial synchronization between the involved courses can be achieved, e.g., through the semester teams.

CONCLUSION AND FUTURE WORK

In this article we describe two courses that have been redesigned to include CDIO projects such that they tackle a similar topic from different directions. At the gap between the two courses, the students suddenly develop deep understanding for that topic. The perception of the students towards the changes done in the two courses and the synergy effect generated as a result is largely positive.

We are currently exploring options for enhancing the overlap between the two courses to increase opportunities for students to gain the deep insights described above. We also plan to investigate how other courses can benefit from a similar approach. Since we currently are adapting our study plans again, there is a unique situation to make the necessary changes.

Finally, we want to measure more precisely, how our approach supports learning, and to what extent.

REFERENCES

Appel, A. (2002). *Modern Compiler Implementation in Java* (2nd Edition ed.). Cambridge University Press.

Chu, P. P. (2008). *FPGA Prototyping by VHDL examples – Xilinx Spartan-3 Version* . Wiley.

E. Crawley, J. M. (2007). *Rethinking Engineering Education. The CDIO Approach.* New York: Springer.

Jens Sparsø, T. B. (2011). CDIO projects in DTU's B.Eng. in IT study program. *Proceedings of the 7th International CDIO Conference.*

Patel, Y. N. (2004). *Introduction to Computing Systems: From Bits and Gates to C and Beyond 2/e.* McGraw Hill.

Todirica, E. A. (2014, January). *Homepage of the Course "HW/SW Programming"*. Retrieved February 2014 from DTU Compute: http://www.compute.dtu.dk/courses/02321/

BIOGRAPHICAL INFORMATION

Edward Alexandru Todirica, is lecturing at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark. His current research focuses on embedded systems and digital electronics.

Christian W Probst is an Associate Professor in the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, and director of studies of a B.Eng. study line Software Technology. His current research focuses on organizational security as well as embedded systems and compilers.

Corresponding author

Christian W Probst
Technical University of Denmark
DTU Compute
DK-2800 Kongens Lyngby
+45 45 25 75 12
cwpr@dtu.dk