This work was submitted at the Laboratory of Integrated Analog Circuit and RF Systems of the institute for semiconductor technology of the RWTH Aachen University

Automatisierte Implementierung der Digitalen Konfigurationsschnittstelle für Anwendungsspezifische Integrierte Schaltungen

Automated Implementation of the Digital Configuration Interface for Application Specific Integrated Circuits

Master Thesis

by
Zhihong Lei
Matr.-Nr. 374616

Supervised by: Bastl, Johannes, M.Sc.

1. Examiner: Prof. Dr.-Ing. Stefan Heinen
2. Examiner: Prof. Dr. sc. techn. Renato Negra

M122

Aachen, 12.09.2019

Eidesstattliche Versicherung

Zhihong Lei	374616
Name	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt,
dass ich die vorliegende Master Thesis mit dem Titel

Automatisierte Implementierung der Digitalen Konfigurationsschnittstelle für Anwendungsspezifische Integrierte Schaltungen

Automated Implementation of the Digital Configuration Interface for Application Specific Integrated Circuits

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 12.09.2019	
Ort, Datum	Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 12.09.2019	
Ort, Datum	Unterschrift

Abstract

The different chips developed at the Chair of Integrated Analog Circuits comprise the possibility to configure the different functional blocks of the respective circuit individually. The required control signals are stored in registers, which are accessible for read and write by use of a digital interface like SPI. Due to the growing complexity of the circuits the effort as well as the error-proneness increase with regard to:

- the definition of the registers in VHDL
- the documentation of the registers in LaTeX
- the usage of the register definitions in different programming languages and environments (C, Matlab, etc.)
- the synchronization of all these data sets

In this thesis, we propose a database centered software that automates the implementation of the digital configuration interface for ASIC chips. Based on an existing database design, we developed a suitable data structure. An intuitive operation of the software was achieved by a well designed graphical user interface implemented with Qt, which hides the underlying complexity from users. We paid special attention to data security by establishing a role model that reflects the responsibilities and access permissions within the design team. Besides the data input and management, generation of the VHDL code as well as the register documentation was developed. At the end of the development process, we integrated and tested the software before its delivery to users.

In development of the software we applied software engineering techniques by conducting requirements analysis, software architecture design prior to implementation and testing. A scientific software development process ensured that the software we developed was satisfactory, high quality, and delivered on time.

Acknowledgement

I would first like to thank my thesis supervisor, Johannes Bastl, for his support and understanding before and during the thesis. At the time I contacted Johannes for the thesis I was going to be an exchange student and couldn't start the thesis right away. Johannes had kept this interesting topic for half a year until I returned to Aachen. During the thesis, Johannes also offered me a great deal of support not only regarding the thesis itself but also my personal issues.

I would also like to thank Kazuki Irie, my supervisor when I was a student assistant at the Chair of Computer Science 6. Kazuki introduced the wonderful topic of neural language models to me. When I was looking for an internship, Kazuki also gave me a lot of valuable information and advice.

I would like to thank Simon Wiesler, my mentor when I was an intern at Amazon. It was my first internship and at that time I was a complete novice in the field of speech recognition. Simon gave me strong support, understanding, encouragement and recognition.

Finally, I would like to thank Bing Zhang, Tim Ng and Zhen Huang, my mentors during my internship at Apple, for their support both professionally and personally during and after the internship.

Contents

List of Figures

List of Tables

List of Source Code

1 Introduction

Application-specific integrated circuit (ASIC) is a widely used solution for many electronic applications. In the Chair of Integrated Analog Circuits, a variety of ASIC chips are developed. The architecture of those ASIC chips can be illustrated by Figure 1.1. They consist of different functional system blocks, whose design and implementation can be performed independently by a corresponding designer. In practice, the system blocks must be configurable individually, so that the chips are flexible enough to meet various requirements and work in different situations. The system blocks are configured by predefined signals, which are stored in registers. To configure the chip, we write specific values to corresponding registers via a digital interface such as SPI. Status information of the system blocks is also buffered in registers and is accessible via the digital interface.

To implement such a digital configuration interface, the designers are required to

- Define the control and information signals in each system block.
- Define registers and map them to corresponding signals.
- Write documentation of the control and information signals and registers.

The digital configuration interface is implemented in a Hardware Description Language such as VHDL and the documentation is usually written in LaTeX. Each part of the interface and documentation are written by a corresponding system block designer. Finally, there has to be a single designer called a **chip owner**, that collects all the VHDL and LaTeX source code and make up a complete SPI interface and documentation.

There are, however, several shortcomings in this manual design approach. First, implementation of the documentation and the interface can be very tedious and labor intensive.

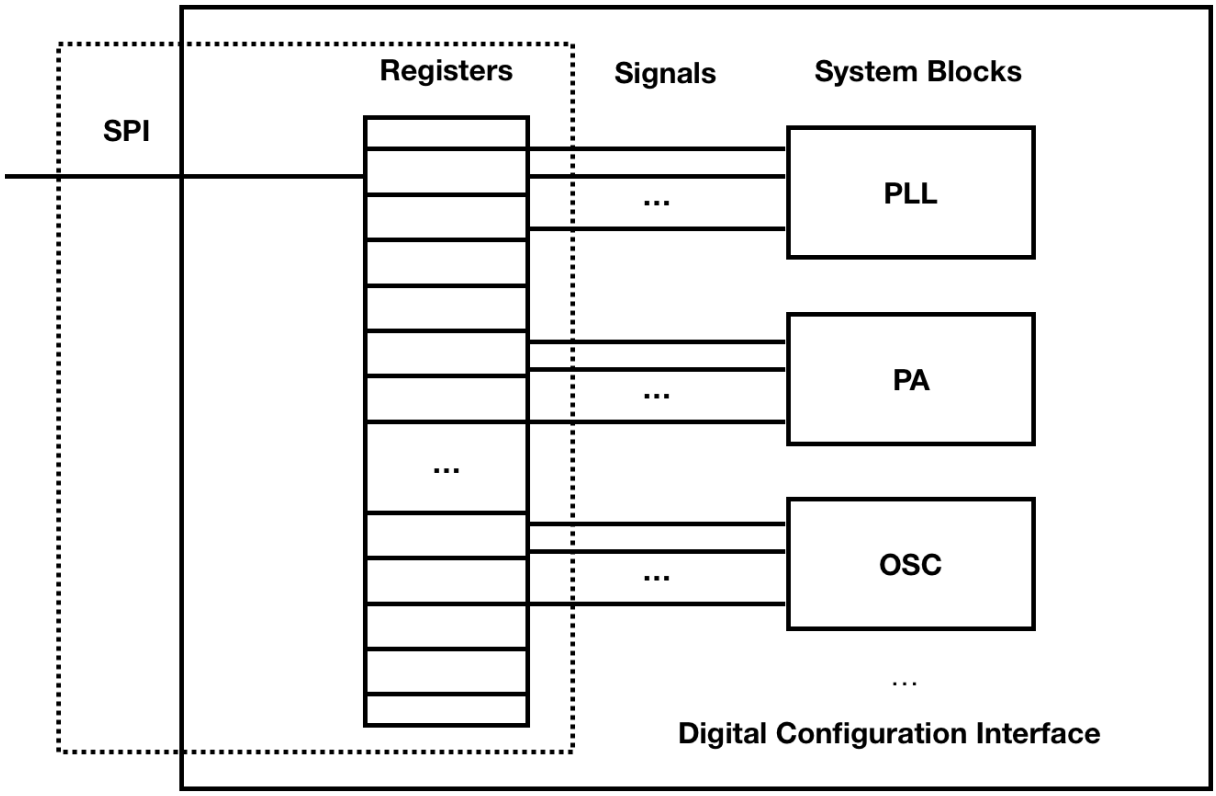


Figure 1.1: Structure of an ASIC chip

The designers have to write everything completely by hand. However, much of the source code has a strong pattern and can actually be generated by software.

Then, manual implementation of the digital configuration interface can be very error-prone. A good example would be generation of initial values for registers. For each register bit, users have to find out which bit of which signal it is mapped to, and then convert the initial value of the signal to binary, and get the value of the corresponding signal bit, that is the initial value of this register bit. Some register bits might not be mapped to any signals, then its initial value could be just zero. Finally, initial values of each register bit are concatenated and the result is converted to hexadecimal. The designers have to be very careful not to map a register bit to a wrong signal bit, or write down a wrong value for a register bit.

Also, special attention must be paid to synchronization of the VHDL source code and the documentation. For example, when the initial value of a register is updated, the designers

have to update the same value everywhere.

Finally, the chip owner has to communicate well with each system block designer. This requires additional efforts, and might cause errors resulted from poor synchronization or communication.

In practice, the chips designed by the Chair of IAS can contain tens of system blocks with hundreds of registers, and even more signals. As the complexity of the chips grows, the shortcomings become a serious problem.

Against this background, we propose a software shown in Figure 1.2 that automates the implementation of the digital configuration interface. The basic idea is simple: the system block designers define the registers and signals and signal-register mappings using our software. They put in document items for each system block, signal or register using a friendly GUI. When the definition is finished, the software can collect all information about the chip, put everything together and generate the VHDL source code and a complete documentation automatically.

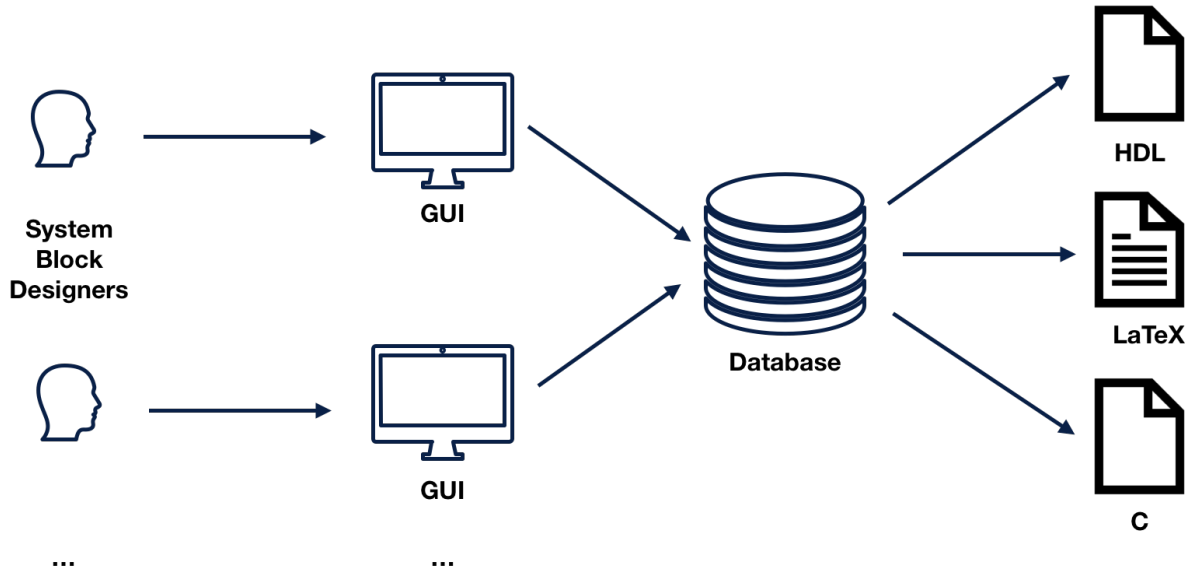


Figure 1.2: Proposed Software Solution

Every designer works independently on a single system block, but to generate a complete VHDL source code and documentation, information about all system blocks has to be

collected. Due to this fact the software must use a shared data storage which is accessible to all users. We propose to use an open source SQL database system as our data container. With an appropriate database design, we are able to manage the data in a highly secure and efficient way.

The proposed solution has the following great advantages

- Designers only have to do minimum necessary work, such as defining the name and type of the registers and signals, defining initial values of the signals, mapping signal and register bits and so on. The rest of work, such as inferring initial values for registers, will be done by the software. By developing a highly friendly user interface, human labor can be reduced to a minimum level. Finally, when definitions of all system blocks are complete, the chip owner does not have to manually collect information. The software shall retrieve all required information from the database and generate the VHDL source code and documentation.
- Synchronization of different outputs is intrinsically ensured. The shared database is the unique data source. Whatever is changed, will be directly reflected on all exports.
- Errors are less likely to occur, since the most error-prone parts are taken care of by the software. In fact, if all information given by the designers is correct, errors can ideally be eliminated. This requires a highly reliable software, though.

2 Software Development Model

We identify our project as a software engineering task. To deliver a high quality software product in time, we have to study different development process models, and adopt an appropriate one. The software development model is an abstract process of how software is developed. As software engineering evolves, many software development models have been introduced. Despite of their differences, basically all these models involve the process of requirements analysis, architecture design, implementation, testing etc. We acquired the knowledge of the whole software engineering process primarily by studying [1].

2.1 Waterfall Model

The waterfall model is a preliminary software development model. It breaks down the software development process into linear sequential phases. Each phase only depends on the output of the previous phase. The developers have to work phase by phase, and there is no way to turn back.

The waterfall model is illustrated in Figure 2.1. The first phase is requirements analysis. The developers use different methods to abstract user requirements, and make the system specifications. Requirements analysis is not as easy as it seems to be, because there are many practical issues. In many cases, the developers do not have good access to users, making it difficult to analyze users' requirements. In fact, even the users themselves might not really know what they want. In some cases, due to lack of software engineering practice, users or clients might come up with unrealistic requirements. When the user requirements are available, the developers have to define the system specifications, aka system requirements, which are a more detailed description of the software system. Finally,

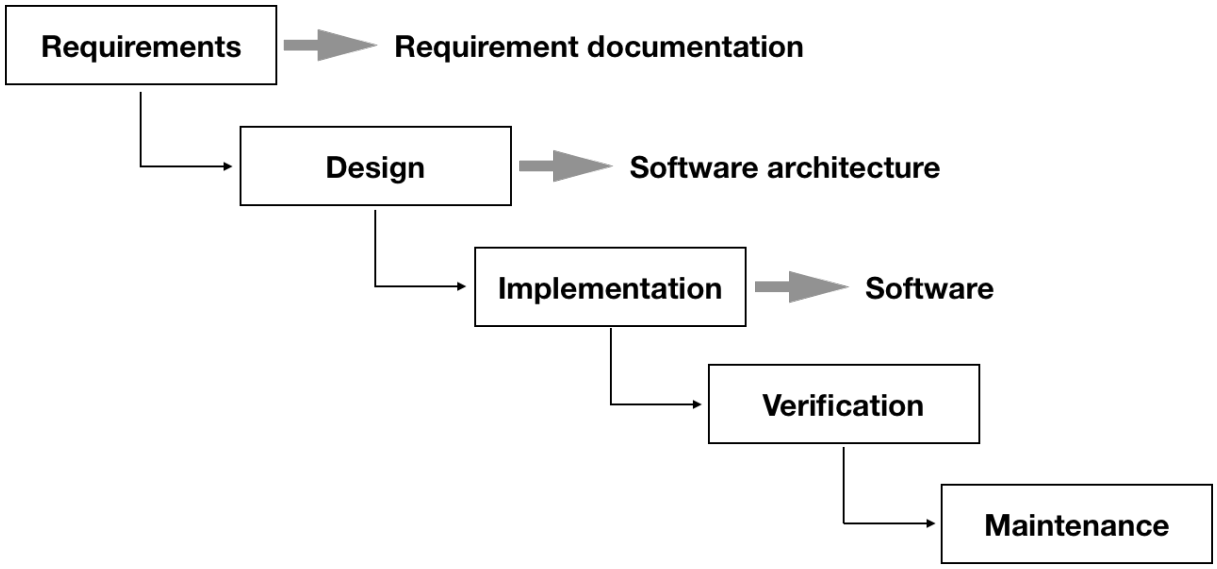


Figure 2.1: Waterfall Model

the output is a requirement documentation either in natural language or formulated language.

With the requirement documentation available, the developers can start designing the software architecture. The design process is crucial because the software architecture is difficult and costly to change in the future steps. A bad architecture can lead to poor testability, maintainability and quality. It can also make implementation of the software more difficult.

When the architecture design is complete, the developers start to implement the software. Then, the developers must test the software and verify whether all requirements are satisfied. Finally, the software is deployed and maintained in everyday use.

The waterfall model breaks down the software development process in a highly structured way. It pays attention to the early stages of the software development process, which tends to be neglected by many developers. According to studies [2], time spent at the early stages can lead to a great cost reduction in the later stages. A major drawback is inflexibility. The waterfall development process does not provide any way to adapt to changes in requirements. Also, there is no way to look back and make changes on the previous steps.

2.2 V-Model

The V-model is an extension of the waterfall model. As the Figure 2.2 shows, the V-model breaks down the development process into the project definition and the testing and integration phases. Each development phase has an associated testing phase.

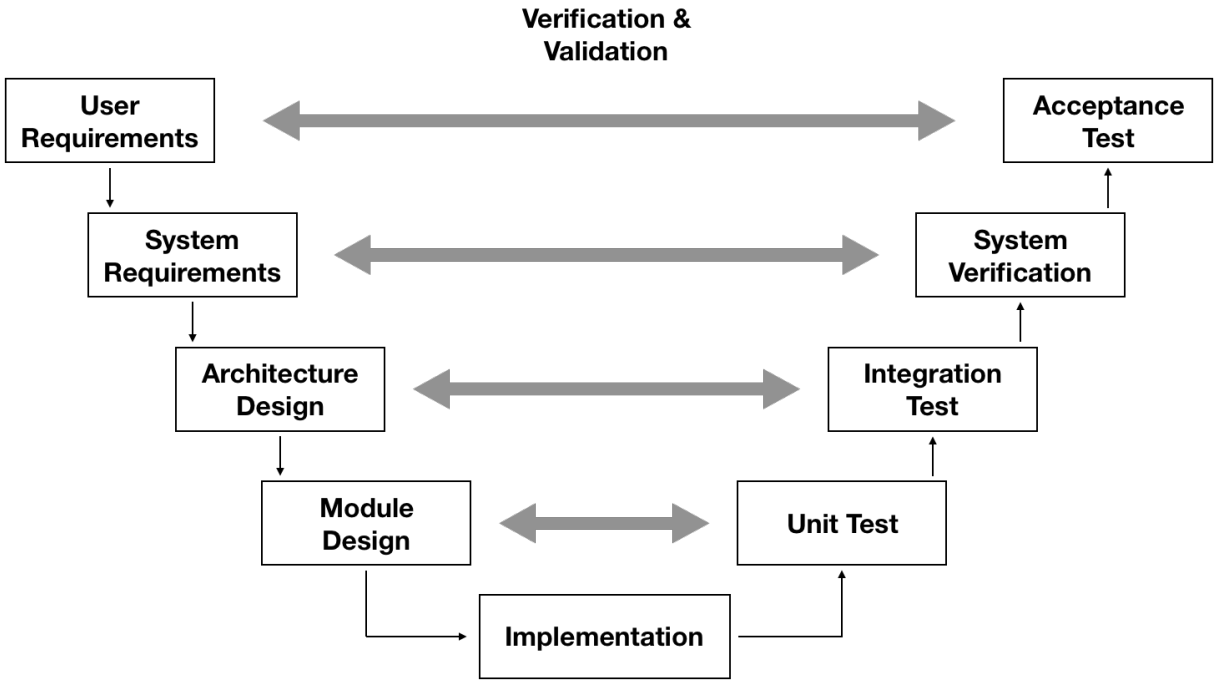


Figure 2.2: V Model

The V-model puts emphasis on testing. However, it receives similar criticisms as the waterfall model. The process is rigid and unable to respond to changes.

2.3 Iterative Model

To tackle problems with the waterfall model and the V-model and to introduce more flexibility, the iterative model was proposed. The idea is to repeat the whole development process and develop the software iteratively, as shown in Figure 2.3. After each iteration, the software is delivered to the users for evaluation and temporary use. Then, the

developers work with the users and update the requirements. After that, the developers repeat the design, implementation and testing phases, and deliver an updated software to the users. Developers repeat this process until the users are satisfied, and then make the final deployable version.

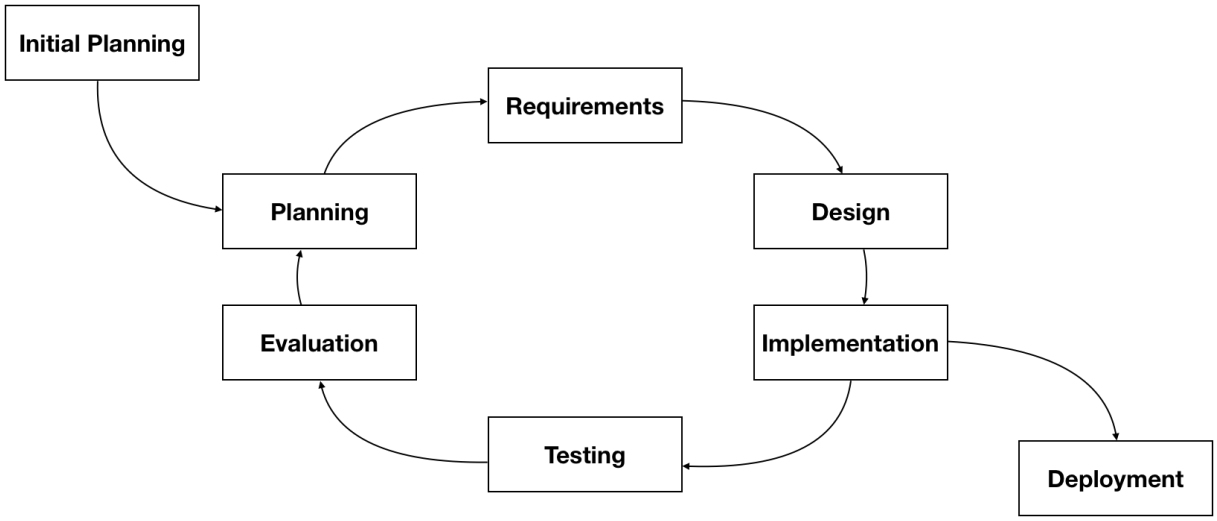


Figure 2.3: Iterative Model

The iterative model differs from the waterfall model in many important ways. In the waterfall model, requirements, software design and implementation are fixed once they are finished. Users do not participate in the development process once requirements are fixed. The software is delivered only when the whole development process is complete. Finally, the waterfall model is only suitable for large projects. The iterative model, however, assumes everything is variable in the first place. After every iteration, users are involved in evaluation of the software. The software is developed incrementally, and can be delivered at early iterations. The iterative model is especially suitable for small projects, however, it can also be beneficial to large projects.

2.4 Prototype Model

The waterfall model assumes requirements can be completely abstracted at the early stage of software development, and remain unchanged afterwards. In practice, however, initial

requirements might not be complete and they can change over time. As is shown in Figure 2.4, the software developers build prototypes, review them with users and update the requirements. In this way, users are involved early in the project and provide valuable feedback to the software developers. Prototyping is also a technique to verify whether the designed technical specifications are fulfilled. Based on existing prototypes, current implementation can be improved and prototypes evolve into a final product. Additionally, prototyping is also a good way for developers to get an initial idea of the project and estimate timelines of the project.

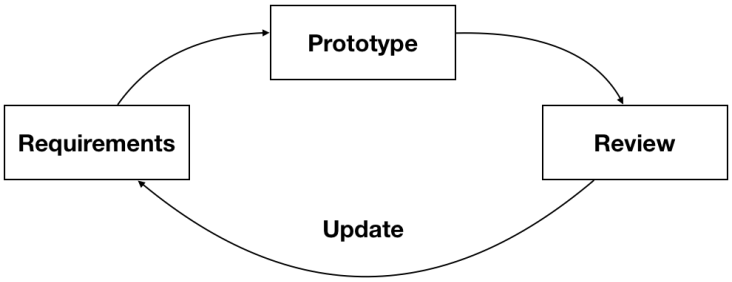


Figure 2.4: Prototype Model

In the field of software engineering, a variant of the prototype model known as rapid prototyping is especially good for designing the user interface. It is not easy to know what the UI should exactly look like without building one. The developers first write preliminary requirements and a draft of the UI. Then, a prototype is built against the requirements. The prototype is reviewed by the users, and if necessary, the requirements are updated and the prototype is revised. Finally, the requirements are fixed.

Rapid prototyping is not concerned about implementation of complete functionalities, but only about some certain aspects like UI. Also, prototypes are made for review, and are thrown away afterwards. Sometimes rapid prototyping is also referred to as throwaway prototyping.

2.5 Conclusion

We investigated different classical software development models. In recent years, the agile development model and its variations such as SCRUM [3] have become increasingly popular.

It emphasizes adaptive planning, face-to-face communication, incremental development and rapid delivery, adaptation to change, and collaboration between the development teams and users etc. It is a totally new development philosophy compared to the classical development models.

In our software development practice, we did not rigidly adopt a certain development model. To better understand the project, develop the final version of the requirements and design the UI, we did rapid prototyping. In the following stages, we adopted an iterative and incremental strategy. We set up a weekly meeting to discuss the newly implemented functionality and the current development status. After the first implementation was complete, we delivered a beta version to users. According to user feedbacks, we removed errors and made minor changes to the software. And finally, we made a product version and delivered it to users.

In the whole development process, users actively participated. We responded to user feedbacks rapidly and adapted very well to changes in requirements. A scientific development process ensured that the software product was satisfactory, high quality, and delivered on time.

3 Requirements Analysis

In this chapter, we will discuss the process of requirements analysis. Requirements have two aspects: user requirements and system requirements. User requirements reflect what kind of service the software is expected to provide to its users. They can usually be an abstract description of the software. System requirements are more detailed descriptions of the functionalities, services and constraints of the software system. Requirements analysis is usually the very first step of software engineering. It is that important because it will be very difficult to change the requirements in the next stages of software development. Failure in requirements analysis can lead to the consequence that requirements cannot be satisfied or the software cannot be delivered in time.

There are various approaches to requirements analysis [4]. Since the requirements for our software are comparably simple and users are well accessible, we decided to adopt the interview approach. The interviewee was the supervisor of the thesis, who would also be a user and admin of the software. The interviews were either in form of emails or face-to-face talks. We had several rounds of interviews. After each interview, we updated the requirements and raised new questions. We did this in an iterative way until we confirmed the requirements were complete. We might want to add more functionalities in the later development stages. But no major modifications would be done on the requirements. We formulated the requirements with natural language.

3.1 User Requirements

To make an easy-to-use software, we have to design a very friendly graphical user interface. Different UI components have to be well organized. They should be simple and intuitive,

with underlying logic hidden from the users. One of our goals is to minimize users' operations on the GUI.

The goal of the project is to develop a graphical user interface based software that aids in implementation of the digital configuration interface for AISC chips. It shall allow chip designers to define the system blocks, different types of registers and signals, initial values of signals, and signal-register mappings, and to write different document items of different kinds, including pure text, mathematics equations, tables and images, on different hierarchy levels using a friendly dialog. The software shall be able to display defined system blocks, registers, signals, signal-register mappings, chip designers, register pages and document items etc. in a navigable way. After definition of the chip is complete, the software shall be able to generate working VHDL and LaTeX source code. The user interface shall be carefully designed and hide as much details as possible from the chip designers. To implement the GUI, an appropriate framework should be selected.

The software shall allow chip designers to work simultaneously on different system blocks of the same chip. All inputs given from different users shall be stored in the same well-designed SQL database running on an appropriate database management system.

For data security, user access control and a login mechanism shall be implemented. A basic role model shall be designed, such that standard users are distinguished from the admins in both database or software level and project level permissions. To make the software more reliable, sanity checks must be implemented especially before database operations. A logger shall be implemented, and user activities and database operations shall be logged.

3.2 System Requirements

3.2.1 Main Window

- The central component of the GUI is a main window.
- The main window shall contain two components: the navigator and the working area.

- The navigator shall be a tree view in a CHIP-BLOCK-REGISTER-SIGNAL structure. It shall allow users to search for components of a given pattern and highlight the matching components. The working area shall contain a chip editor on the left and a document editor on the right. The two editors can be switched on or off.

- The chip editor view shall be a stacked widget containing two pages: the chip level page and the block level page. The chip level page shall display read-only information about the basics of the chip, system blocks, chip designers and register pages. The block level page of the chip editor shall contain two tabs: one for registers and the other for signals. On the signal tab there should be two tables showing signals of the current system block and signal-register mappings of the current signal. The register tab is organized in a similar way.

- The document editor view shall contain a table showing the list of document items, and a document editing area. The editing area is hidden. When users add a document item or edit an existing one, the editing area shall be displayed.

- The working area shall respond to changes in the current item in the navigator. When the current item changes, the chip editor shall switch to either the chip level or the block level page. If in the block level page, either the register or signal tab is set as current. Then, the corresponding information shall be displayed on the chip editor. At the same time, the documents regarding this item shall be displayed, if the document editor page is switched on.

- Unless otherwise noted, below each table in the chip editor and the document editor, there shall be an **Add** and a **Remove** button. The users shall be allowed to add a new item, remove or edit an existing item.

- At the top of the main window should be a menu bar containing actions on users, chips, views, exports and so on.

3.2.2 Chips and Documents

- The software shall allow users to edit chips and documents in dialogs. Each dialog is designed for a specific use case or functionality, such as adding a new system block. The dialogs must provide appropriate components for users to put in information.
- The software shall allow users to add new chips or edit exiting chips. It should allow users to put in the chip name, register width, address width, and select whether the chip is MSB first.
- The software shall allow users to add new system blocks or edit existing ones. It should allow users to put in the block name, block abbreviation, start address in hexadecimal format and select the responsible designer. The start address should fit the chip's address width.
- The software shall allow project admins to add new chip designers or edit existing ones. It should allow admins to select a designer out of all users in the database, and select the designer's project role.
- The software shall allow users to add new registers to the current system block or edit existing ones. It should allow users to put in the register name and select a register type out of all register types in the database.
- The software shall allow users to add new signals to the current system block or edit existing ones. It should allow users to put in the signal name, width, select the signal type, and check whether to add a port for this signal. If the signal is a register signal, i.e. it shall be mapped to registers, the dialog shall allow users to select a register type. When the register type is writable, it shall also allow users to define the initial value in hexadecimal format.
- The software shall allow users to map signal bits and register bits in an efficient and convenient way. A signal bit can only be mapped to one register bit, and the other way around.
- The software shall allow users to add documents to the currently selected item in the navigator or edit existing ones. It shall allow users to select a document type. For

a text document, it shall allow users to put in text data. For an image document, it shall allow users to select an image and edit its caption and width. For a table document, it shall allow users to edit the table contents in a table widget and put in its caption. The number of rows and columns of the table shall be adjustable.

- The software shall allow users to add register pages or edit existing ones. It shall allow users to select an available control signal and specify the page name and a valid page count. It shall allow users to select available registers in a convenient way. The control signal must not be mapped to any registers in the page it controls. A signal can control at most one register page block.

3.2.3 User Access and Authentication

- The software shall allow database admins to create users conveniently.
- A login mechanism shall be implemented. The main window of the software shall open only when the users provide correct username and password through the login dialog.
- The software shall allow users to select an existing chip to open. The chips shall be listed in a table containing basic information about them.
- Each user shall be assigned a database role. The database roles shall be defined in the database, determining whether the users can or cannot add or remove projects or users.
- Each project shall have at least one chip designers. Chip designers shall be assigned a project role. The project roles are defined in the database, determining whether the users can add or remove system blocks, chip designers, register pages, registers, signals, signal-register mappings, and documents belonging to a specific chip item (i.e. a system block, a register etc).
- The UI shall be adjusted according to the the database role and the project role. For example, some widgets might be disabled if the user is not eligible for it.

- When the design of a certain chip is complete, the chip owner can freeze the chip. In this case, everything about the chip, except documentation, is frozen, and cannot be edited anymore. In such case, the UI shall disable relevant UI components. It shall allow the chip owner to unfreeze the frozen chip.

3.2.4 SPI Interface and Documentation generation

- The software shall be able to generate a VHDL SPI interface. It shall allow users to select SPI interface templates and configure the output before generation.

- The software shall be able to generate a LaTeX documentation. It shall allow users to configure the documentation, for example, to select which system blocks to include in the documentation.

3.2.5 Reliability

- Error handling for database operations shall be implemented.

- All write operations on the database and certain user activities shall be logged for error diagnosis purpose.

- The software must check if each chip editing or document editing operation, such as adding a new system block or removing a register, is legal. The software shall check e.g. names of the signals/registers/system blocks/users etc, the initial value of a signal, the start address of a system block and so on.

- Before generating the VHDL source code, the software shall check whether the configuration file contains all necessary predefined variables, whether address blocks of different system blocks overlap or exceed the address width, and whether the signal-register mappings are valid.

3.3 Conclusion

Requirement engineering is the very first step of software development. It is of crucial importance, since it is very costly to change the requirements in the following stages. In case of a comparatively small software project, we might be able to define all requirements before the following steps. However, when the project becomes more complex, it will be difficult to write a complete requirement list. There are cases where new requirements have to be added and existing requirements have to be modified.

In our practice, we first made a draft containing the most fundamental requirements. We then made a prototype to help us determine requirements in particular regarding the UI. Other requirements regarding data security and reliability can be comparably easily determined without a prototype.

4 Selection of Infrastructure

In the software engineering practice, it is a common case that we depend on external tools and infrastructure, because we cannot build everything from scratch. To develop a software with a graphical user interface, we have to use a GUI framework. In our project, we also have to use a SQL database management system. They are the infrastructure of our software solution. Before implementation, we have to decide what infrastructure we want to use. To select the best infrastructure a few aspects must be taken into account.

We should consider the license, which stipulates the conditions on which we could use the software. There are many open source GUI frameworks and database systems. Their license could be tricky, however. Some open source licenses require that the users shall open source their software if the source code of the toolkit is used. This practice is referred to by the community as *copyleft* [5], since it is against copyright. We want to preserve copyright of our software as much as possible, so we would prefer those toolkits bearing a looser license.

Another aspect is supported operation systems or platforms. When we select the infrastructure, we would prefer those cross-platform toolkits.

Resources such as learning materials and documentations are also of major importance. Some toolkits have really good official documentations on use cases, sample code and descriptions of the APIs, but others not. These documentations can greatly help us learn the toolkit and use it in our software. Apart from the official resources, the community also plays an important role. It is much easier to get a solution from a larger community when we run into problems.

For the GUI framework, we should consider the supported programming languages. Although programming language is not a high priority, we would prefer to user a modern,

friendly language. Some GUI frameworks might offer a specialized designer which makes UI design much easier. The API should also be taken into account. We would definitely prefer a friendly and concise API.

4.1 GUI Framework

Many GUI frameworks are developed and used to build software applications. We investigated and selected the most popular listed Table 4.1.

Table 4.1: Popular GUI Frameworks

Toolkit Name	Cross-platform	Programming Language	Specialized IDE	License
Qt	Yes	Multiple	Yes	LGPL
MFC	No	C++	No	Proprietary
GTK	Yes	Multiple	Yes	LGPL
Swing	Yes	Java	Yes	-
Tk	Yes	Multiple	No	BSD
wxWidgets	Yes	Multiple	No	WxWindows license

The first framework we will discuss is Qt. It is also the one we finally chose. It is one of the most popular open source GUI frameworks. The Qt framework is written in C++. However, the Qt company and community developed wrappers for various programming languages including the popular Python. Qt is distributed under the LGPL license. It is a fairly loose license under which we can use Qt for free and do not have to open source our software. Because of this, it is possible to commercialize our software. The Qt company developed a highly customized IDE, the Qt Creator, for developing Qt applications. With Qt Creator, we can design GUI in a what-you-see-is-what-you-get (WYSIWYG) manner.

GTK, Tk and wxWidgets are also very popular cross-platform GUI frameworks. They are all written in C or C++. However, like Qt, they also provide wrappers for other languages. Despite of some differences, their licenses are all pretty loose. However, one shortcoming compared to Qt is they do not have a specialized IDE, although we might use a third-party one. Another shortcoming is they have less documentation. Also, unlike Qt, which is

supported by the Qt company and is partially commercial, they are totally maintained by the community. The developer community is also smaller than Qt's.

Microsoft Foundation Class (MFC) is a C++ library for developing desktop applications on Windows operating system. It is essentially an object-oriented wrapper for the Windows API. Due to this fact, it only supports Windows operating system and is not cross-platform. Microsoft offers a community version of Visual Studio, which is recognized by many software engineers as the best IDE for general C++ programming, and it supports MFC very well. Users can design the GUI in a WYSIWYG manner. But MFC is proprietary. Also, it is a little bit out of date. In these considerations, MFC is not a good choice for us.

In the end, we decided to select Qt as our GUI framework. Although PyQt [6], the Python wrapper, is being very popular nowadays, we still decided to use the C++ API for several reasons. First, Qt Creator does not support PyQt very well. Second, we can build a standalone application in Qt C++. If we use PyQt, however, we have to either depend on Python, or compile it into an executable, which requires additional effort and might bring us more uncertainty or unreliability. Finally, C++ can be much faster than Python, which is beneficial for a GUI application. In fact, the Qt C++ API is very concise and elegant, so we could still develop the software pretty fast. Also, most documentations of Qt are about the original C++ API. Considering all these points, we chose C++ as our programming language.

4.2 Database Management System

The database is crucial for our software. A shared database ensures data consistency and makes cooperation on the same chip project easier. The database we use shall fulfill the following requirements:

- Accessible to all users.
- Can be concurrently read or written by multiple users.
- Data must be reliably stored.

- Data should be well organized.
- Data access should be efficient.

A relational database is comprised of tables with a certain number of columns. Each column is a data field. It is called relational because a table column can refer to a column in another table. This is one of the core ideas of the relational database. With a relational database, relationship between data can be easily and efficiently handled. The database held on a relational database system is managed with SQL, the Structured Query Language. Thus, such a relational database is often referred to as an SQL database.

The SQL database has many benefits. First of all, it supports highly efficient queries. Usually each table has at least one index field. With the index, the performance of queries can be greatly sped up. Another benefit is concurrency. The database management system is designed for concurrent queries from the very beginning. It can also take care of concurrent writing operations. The database system usually implements an account mechanism, and the data is not directly exposed to users. In this way, data can be securely and reliably stored. Besides that, many database management systems adopt the client/server model. The database can be held remotely on a host machine and users can access it from their local machines. Our software requires a shared database. Although we can put the database on a shared file system, a database server would provide much more flexibility.

Table 4.2: Popular Database Systems

Toolkit Name	Cross-platform	License	Specialized IDE	Comment
MySQL	Yes	GPL	Yes	Most widely used open source database system
Microsoft Access	No	Proprietary	No	Expensive
SQLite	Yes	Public domain	No	Limited functionalities
Microsoft SQL Server	No	Proprietary	Yes	Expensive
Oracle	Yes	Proprietary	Yes	Expensive
MariaDB	Yes	GPL	Yes	Compatible to MySQL

Among the database systems listed in Table 4.2, Microsoft Access and Microsoft SQL Server are famous systems on Windows operating system. They are both proprietary software. Microsoft SQL Server is of the client/server model while Microsoft Access is not. Oracle is one of the most reputable commercial database management systems. It is proprietary and quite expensive. SQLite, as the name indicates, is a lightweight SQL database system. It provides limited functionalities and is designed for holding small-scale data locally. In fact, Qt itself includes the SQLite database system.

We are especially interested in MySQL [7]. It is the most popular open source database in the world. It is released under the GNU General Public License (GPL) version 2, or the proprietary license. MySQL is used many popular websites including Facebook, Twitter and Youtube. It is a component of the famous LAMP (Linux, Apache, MySQL, PHP) web application software stack. After MySQL's acquisition by the Oracle company, the original developers forked MariaDB [8] from MySQL. It is basically completely compatible to MySQL.

Besides the differences mentioned above, these SQL database systems also have some discrepancy in the SQL language. It is not a problem though. In the end, we chose MySQL or MariaDB in our project. For the software to access the database, we need a MySQL connector which can be downloaded from the official website. It is to be noted that MySQL and MariaDB use the GPL license where copyleft is present. However, the database connector is released under the LGPL license and does not require copyleft. In our software we only need to incorporate the connector, and the database management system is standalone, so we do not have to open source our software.

5 Software Architecture Design

In this chapter, we will discuss the process of software architecture design. It is a crucial step of software engineering. A poor architecture might often result in low quality, reliability and maintainability. It also makes development more difficult. The architecture is hard to develop in an incremental way, and it is costly to change in the later stages. Therefore, enough attention should be paid to the architecture design.

5.1 Layered Software Architecture

There are different types of software architectures, such as layered architecture, event-driven architecture, micro-kernel architecture, micro-services architecture, cloud architecture etc. [9]. The layered architecture is the most widely used, and the de facto standard software architecture.

The layered architecture contains roughly four layers: the presentation layer, the business layer, the persistence layer, and the database layer. The presentation layer is where the users interact with the software. It displays internal status of the software and receives user inputs. In case of a GUI software, the presentation layer consists of GUI widgets. The business layer is where a certain business logic is defined and implemented. The persistence layer, also known as data layer, is where the users access the database layer. The database connector API, for example, is on this layer. The bottom layer is the database layer containing SQL databases, files, and so on.

Each layer contains multiple modules that achieve a specific purpose. Within a specific layer, modules are fairly independent of each other. For example, we might want to define

a dialog for adding new system blocks and another for adding registers. These two dialogs basically have nothing to do with each other. However, in practice, modules in a certain layer might also need to directly interact with non-neighboring layers. A well-designed layered architecture allows users to easily revise a certain module, or add a new module into a certain layer, without much modification on other modules on the same layer or neighboring layers.

5.2 Layered Architecture Design

With the layered architecture, what we need to do is divide the software into function modules and fill them into those layers. To design a good layered architecture, it is crucial to define good modules. We have to ask ourselves these questions:

- What modules should be there?
- What is the boundary of each function module?

We found it might be a good idea to define modules by functionality. For each functionality, we create a UI module and a business logic module. We put them in the presentation layer and business layer respectively. The persistence layer and the database layer in our case is quite straightforward. Intuitively, user authentication shall be somewhere between the presentation layer and the business layer. When users do something on the GUI, the authenticator shall check whether the operation is legal, and determine whether to execute the corresponding business logic function or prompt an error message to the user. In fact, we also want the presentation layer to adjust its appearance according to the authentication. As a result, we decided we should modify the default four-layered architecture by adding an authentication layer between the presentation and the business layer.

Following what is discussed, we listed functionalities from the requirements. For many of those functionalities a certain UI component is required. Some functionalities might share a single UI component. For example, when we add or edit a system block, we need the

same dialog. Finally, we designed the software architecture as in Figure 5.1. Only part of the modules are included in the figure.

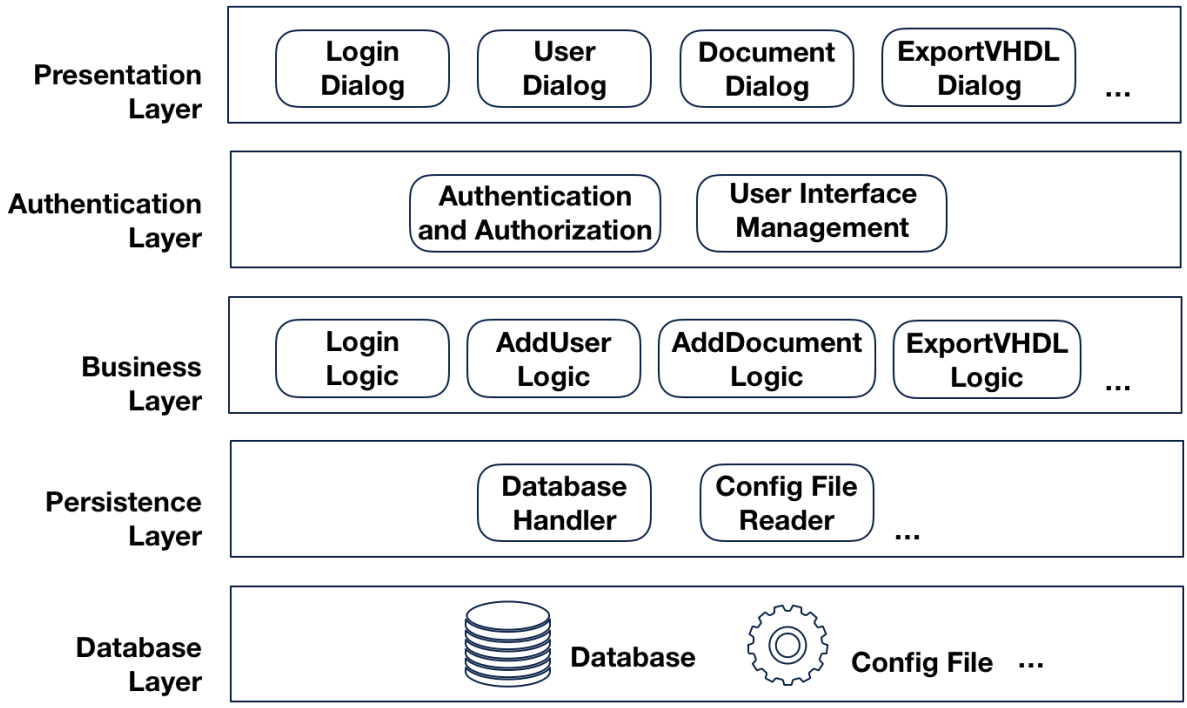


Figure 5.1: Layered Architecture

5.3 Object-Oriented System Design

So far we designed the layered architecture of the software by dividing it into function modules, and assigning each module to a certain layer. The layered architecture, however, does not model the interactions between modules within a layer or across layers. Also, the models are not associated with any implementation form. In practice, they can be implemented with a class or a function. In this section, we will design the software system in more details following the object-oriented programming paradigm, and finally produce a class diagram. However, at this point we do not consider module design and implementation details. We will leave this to Chapter 7.

5.3.1 Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on objects. An object contains data and methods. Objects interact with each other and this usually leads to modification of the data in the objects. In the class-based OOP languages, an object is an instance of a class, a template for creating objects and providing initial values and behaviors.

OOP has three central ideas [10]

- Encapsulation: the classes hide their internal status, and only expose necessary information.
- Inheritance: it represents the *is-a-type-of* relationship between classes.
- Polymorphism: it provides a single interface to entities of different types.

The definition of classes and their relationships can be represented by the UML class diagram. The class is represented with a rectangle with three boxes as in Figure 5.2. The top box contains the class name. The members variables (i.e. the data) are in the

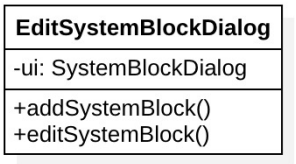


Figure 5.2: Example Class

middle box. In the bottom box are member methods. The member variables and methods are decorated with +, # or - representing public, protected and private respectively. Classes related to others are connected with lines of different forms as in Figure 5.3. Relationships can be categorized into instance-level and class-level. According to the degree of dependency, instance-level relationships are *dependency*, *association*, *aggregation* and *composition*. Dependency relationship exists when an instance references another. Association represents a *has-a* relationship. Aggregation and composition are variants of the

has-a association relationship in that they represent a *is-part-of* relationship. The difference is, the composition relationship entails ownership. If class A has a composition relationship with B, when B is destroyed A will be destroyed too. The class-level relationships reflect the inheritance relationships between classes. The two relationships differ in that realization involves an abstract superclass, aka an interface.

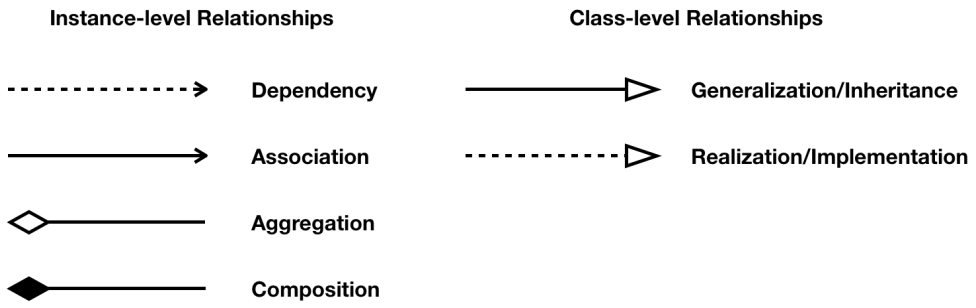


Figure 5.3: Class Relationships

5.3.2 System Design

With the OOP background, our task is to design the system and produce a class diagram. At this point, we have to answer two questions

- What classes should there be and what are their functionalities?
- What are their relationships with each other?

These two questions are not independent. The answer to one can influence the other.

In Qt, a general design pattern is to first design a UI form using the Qt Creator, and then design the class incorporating the UI as one of its member variable. We can design the UI in a WYSIWYG manner. The Qt Creator will then generate a UI form in XML format containing all information about the UI and a C++ class from the UI form. Then, we will create a class with an instance of the UI class. We can then write business logic functions in this class. In this way, modules in the presentation layer and business logic layers are unified.

Some functionalities require user inputs. For example, to add a system block the name, abbreviation and start address of the system block must be given by the user. The client requires developers to design an appropriate dialog for such use cases. However, the dialog can be reused for different functionalities. The system dialog can be used for not only adding but also editing system blocks, for example. In such cases, a single UI class can be associated with multiple functionalities. Following the same pattern, we can design dialog classes for other functionalities such as adding or editing a signal, register and so on. Since these functionalities are independent of each other, these classes basically do not have relationships with each other, and we can design and implement them in an incremental manner.

According to the requirements, the main window shall contain a chip navigator, a chip editor view and a document editor view. Intuitively, we can design a **ChipNavigator**, **ChipEditorView** and **DocumentEditorView** class respectively. The **ChipEditorView** displays information about the chip and provides access to those functional dialogs related to chip editing. Similarly, the **DocumentEditorView** displays document items and provides access to document editing dialogs. The **MainWindow** itself is also associated with global level functional dialogs such as **SPIGenerationDialog**, **DocumentationGenerationDialog**, **ChangePasswordDialog** and so on. In the end, the system can be illustrated with the class diagram as shown in Figre 5.4.

Note that this class diagram is simplified in many ways. First, it only shows classes we want to create. The classes provided by Qt are omitted. The UI classes are not in the diagram either. Instead, they appear as a member named *ui* of many classes in the diagram. Since we were still in the design phase so the classes were incomplete. Only the most important member variables and methods are shown in the diagram. Also, most classes have a dependency relationship with the **DatabaseHandler** class. However, their relationships are omitted in the diagram so as to make the diagram more readable and understandable.

The center of the software is the **MainWindow**. From the diagram we see that the **MainWindow** *has* a **ChipNavigator**, **ChipEditorView**, **DocumentEditor**, **Authenticator** and a **LoginDialog**. Unlike other dialog classes such as **CreateUserDialog**, the **LoginDialog** is a member of the class **MainWindow** and it exits in the whole lifetime

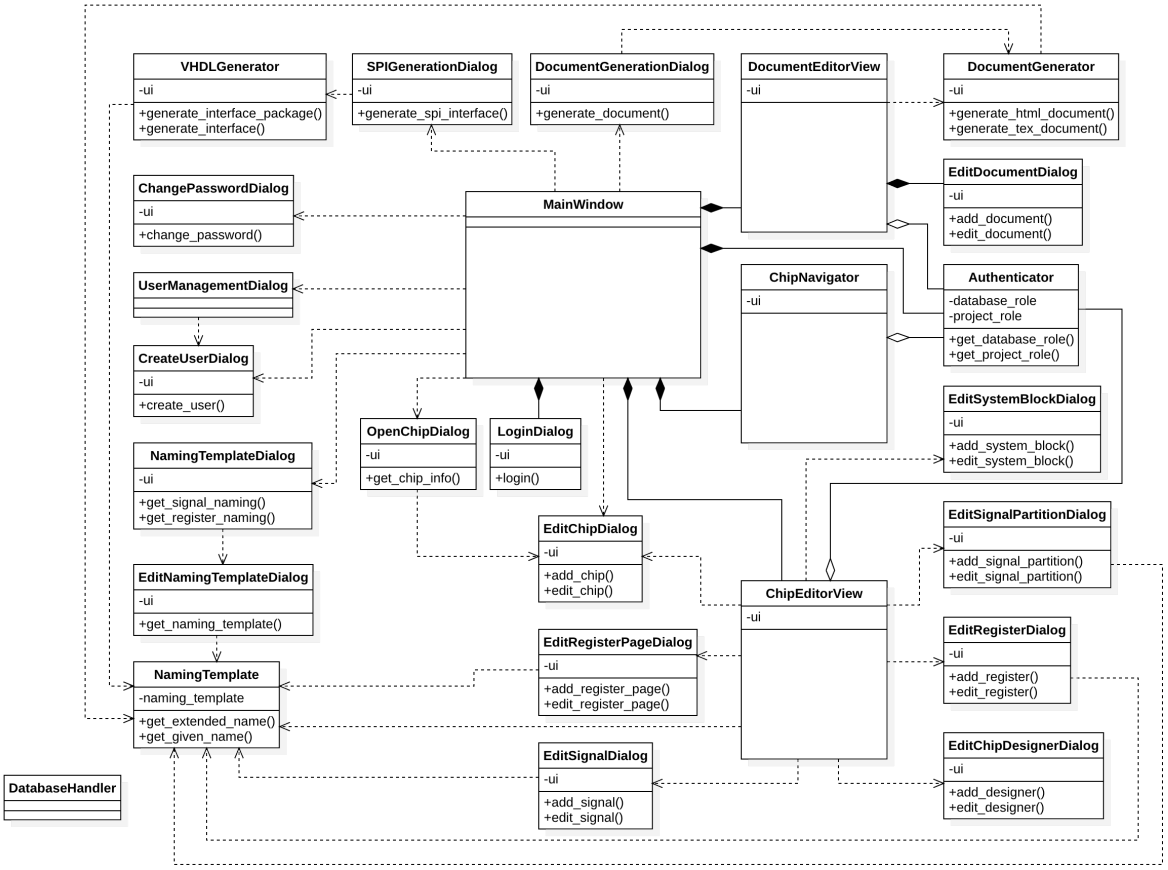


Figure 5.4: Register Manager Class Diagram

of the software (which is exactly the lifetime of the **MainWindow**). This is because we consider that users may want to log out and log in with another account. The **MainWindow** is directly dependent of many dialogs which are not directly associated to the **DocumentEditorView** or the **ChipEditorView**. These dialogs are not members of the class **MainWindow**. When they are needed, a certain action is triggered by the user and a temporary instance will be created.

The **ChipNavigator**, **ChipEditorView** and **DocumentEditorView** are members of the **MainWindow** and they exist in the whole life cycle of the software. The **ChipEditorView** are dependent of those chip editor dialogs including **EditChipDialog**, **EditSystemBlockDialog**, **EditRegisterDialog**. The **DocumentEditorView** is dependent of

the **DocumentGenerator**. Likewise, these dialogs are created only when needed.

A special case is the **Authenticator**. It is the major component of the authentication layer. It stores the database permissions and project permissions of the current user to the current project. Whenever the current status changes, for example, when the user clicks on another system block on the navigator, permissions are requested and the UI is updated by enabling or disabling certain widgets. The **MainWindow** *has* a member **Authenticator**. The pointer to the member **Authenticator** is passed to the **ChipNavigator**, **ChipEditorView** and **DocumentEditorView**, such that they actually access the same **Authenticator** instance.

Dependency relationships between the **MainWindow**, **ChipEditorView** etc. and the functional dialogs are easy and clear because those functional dialogs are basically independent of each other. However, the relationships between the **ChipNavigator** and the **ChipEditorView** or **DocumentEditorView** are tricky. In our design, they are not directly related to each other, but via the **MainWindow**. Whenever the user clicks on something in the navigator, it will *tell* the **MainWindow** that something is triggered by sending it a **signal** (will be discussed in the Chapter 6). The **MainWindow** then updates the current system block, register, signal etc., and updates the status of the **ChipEditorView** and **DocumentEditorView** by function calls. Likewise, when the chip is edited, the **ChipEditorView** sends a signal to the **MainWindow**. The **MainWindow** then calls a corresponding function to update the navigator.

To generate the SPI interface and the documentation, we use the **SPIGenerationDialog** and the **DocumentGenerationDialog**. With these dialogs, we can specify necessary inputs and configure the export. The **DocumentGenerationDialog** then creates a **DocumentGenerator** and calls the corresponding methods to create documentation rather in LaTeX or HTML (see Section 7.6) format. Likewise, to generate the SPI interface, the **SPIGenerationDialog** creates a certain type of generator given the format of the export.

The **NamingTemplate** is designed in response to the requirement that the names of registers and signals shall be formatted. For example, we may want all register names to start with the block abbreviation and end with REG. To display the current naming

templates, we use the **NamingTemplateDialog**, which also provides entries to editing the naming templates using the **EditNamingTemplateDialog**.

Although the class diagram does not regard how each class is implemented but only what functionalities they provide, we cannot design a software system completely without considering the design and implementation details of each class. We will discuss this in the Chapter 7.

6 Introduction to Qt and SQL Programming

So far we have designed the detailed software architecture. Before starting module design and implementation we have to introduce Qt GUI programming and SQL programming.

6.1 Qt GUI Programming

Qt [11] identifies itself as a cross-platform software development framework. It was originally designed for making graphical user interface applications. The Qt framework consists of the following main modules in Table 6.1 [12]. As the name indicates, Qt Core is the core of the Qt framework. It defines QObject, the base class of all Qt objects. QObject provides lower-level infrastructure for the Qt Framework and applications.

Special attention should be paid to the **Signal and Slot** [13] mechanism supported by QObject. It is a powerful, friendly and intuitive alternative to the callback technique, and is a central feature of Qt. When an event occurs, a signal is emitted. The corresponding slots receive that signal and respond by executing a function. The signal and slot mechanism has three elements: the signal, the slot, and connection between them. They can be illustrated with Figure 6.1.

Qt has predefined many useful signals for the UI widgets, and they are automatically emitted once a corresponding event occurs. For example, when we click on a push button, it will send a **click()** signal. To customize the behavior of an event, we can define slot

Table 6.1: Qt Modules

Module	Description
Qt Core	Core non-graphical classes used by other modules.
Qt GUI	Base classes for graphical user interface (GUI) components. Includes OpenGL.
Qt Multimedia	Classes for audio, video, radio and camera functionality.
Qt Multimedia Widgets	Widget-based classes for implementing multimedia functionality.
Qt Network	Classes to make network programming easier and more portable.
Qt QML	Classes for QML and JavaScript languages.
Qt Quick	A declarative framework for building highly dynamic applications with custom user interfaces.
Qt Quick Controls	Provides lightweight QML types for creating performant user interfaces for desktop, embedded, and mobile devices. These types employ a simple styling architecture and are very efficient.
Qt Quick Dialogs	Types for creating and interacting with system dialogs from a Qt Quick application.
Qt Quick Layouts	Layouts are items that are used to arrange Qt Quick 2 based items in the user interface.
Qt Quick Test	A unit test framework for QML applications, where the test cases are written as JavaScript functions.
Qt SQL	Classes for database integration using SQL.
Qt Test	Classes for unit testing Qt applications and libraries.
Qt Widgets	Classes to extend Qt GUI with C++ widgets.

functions and connect them to the corresponding signal. We can also subclass the standard Qt widgets and define our own signal functions.

In a widget based application, Qt GUI and Qt Widgets must be included. Qt GUI classes provide infrastructure for user interfaces. They are extended to Qt Widgets, which are primary elements for creating GUI. Widgets can display data and status information, receive user inputs, and provide a container for other widgets that should be grouped together. Qt Widgets can be categorized into the following groups

- Layouts are geometry managers. They automatically arrange child widgets in their containers according to their **sizeHint** and **sizePolicy** properties. According to the direction in which the child widgets are distributed, there are vertical, horizontal,

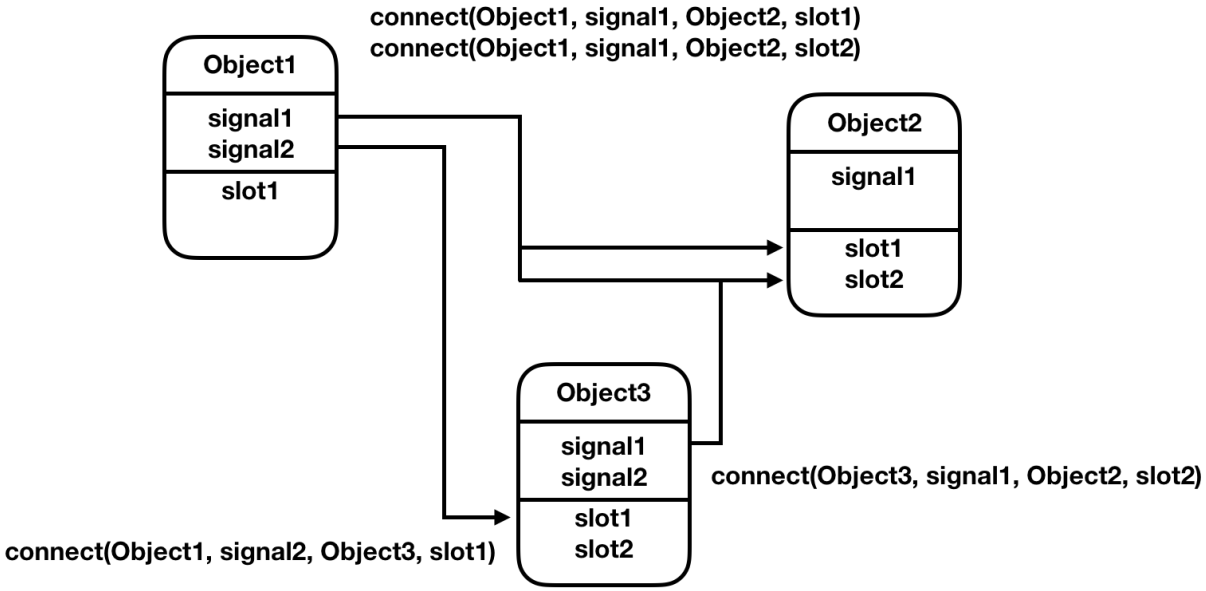


Figure 6.1: Signal and Slot

grid and form layout.

- Spacers provide blank space in a layout. There are vertical and horizontal spacers.

- Buttons provide push button or checkable button widgets, among which the most frequently used are push buttons, check boxes and radio buttons. A special case is the button box, which contains a configurable set of predefined push buttons.

- Item widgets/item views provide architecture to manage the relationship between data and the way it is presented to the user. There are list, tree, table view/widget etc.

- Containers provide an area to group widgets together. There are group box, tool box, tab widget, stacked widget etc.

- Input widgets provide a friendly way to take different types of user inputs. There are combo boxes, line edits, text edits, spin boxes etc. These text widgets can also be configured as readonly and used to display text data. Qt also provides widgets to take date and/or time.

- Display widgets provide a friendly way to display different types of data. There are labels, text browsers, calendar widgets, progress bars and others.

Most of these basic widgets are configurable according to our requirements. We can also customize the widgets by subclassing them. With these widgets, we are able to design a powerful and friendly user interface.

The Qt Company develops an IDE called Qt Creator. It is different from any other IDEs in that it is highly customized for Qt. It provides a very friendly and convenient wizard for creating Qt applications and what it calls designer form classes, which are essentially subclasses of Qt widgets such as **QDialog**, **QMainWindow** etc. or the generic **QWidget**, depending on what kind of UI we want to create. The user interface is generated from the XML UI form and is a member variable of the designer form class. The definition a designer form class is similar to Code 6.1 and 6.2.

Code 6.1: Header of an Example Designer Form Class

```
1  // mainwindow.h
2  #include <QMainWindow>
3
4  namespace Ui {
5  class MainWindow;
6  }
7
8  class MainWindow : public QMainWindow
9  {
10     Q_OBJECT
11 public:
12     explicit MainWindow(QWidget *parent = nullptr);
13     ~MainWindow();
14 private:
15     Ui::MainWindow *ui; // instance of the UI class.
16 };
```

Code 6.2: Source Code of an Example Designer Form Class

```
1  // mainwindow.cpp
2  #include "mainwindow.h"
3  // header generated by Qt Creator.
4  // The UI class is defined here
```

```
5  #include "ui_mainwindow.h"
6
7  MainWindow::MainWindow(QWidget *parent) :
8      QMainWindow(parent),
9      ui(new Ui::MainWindow)
10 {
11     ui->setupUi(this);
12 }
13
14 MainWindow::~MainWindow()
15 {
16     delete ui;
17 }
```

6.2 SQL Programming

In Chapter 4, we have already introduced the relational database management systems and had some basic knowledge about them. We selected MySQL as our database system. Before starting implementation, we have to learn about SQL operations, the MySQL connector and the designed database structure as well.

6.2.1 SQL Operations

SQL operations can be categorized into *database* operations and *data* operations. Database operations allow users to create, remove, update or query databases and tables. To be concluded, they allow users to define a database and manage its data structure. Data operations allow users to insert, remove, update or query data entries in the tables. To develop a database application, we usually predefine a database and its data structure. The database applications only do data operations and do not modify the data structure.

1. Database Operations
To create a database, we first create an empty database in the database management system. After that, we create tables in which data is contained in a structured way. When a table is created, we have to specify the data columns the table should contain and their

corresponding data types. Code 6.3 shows an example of how databases and tables are created.

Code 6.3: Creation of Database and Tables

```
1  CREATE DATABASE ias;
2  USE DATABASE ias; -- use the database ias
3
4  -- create table staff
5  CREATE TABLE IF NOT EXISTS staff (
6     -- data columns
7     staff_id INT AUTO_INCREMENT,
8     name VARCHAR(40) NOT NULL,
9     phone_number VARCHAR(40) NOT NULL,
10    address VARCHAR(100),
11    PRIMARY KEY ( user_id )
12 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
13
14 -- create table project
15 CREATE TABLE IF NOT EXISTS project (
16    -- data columns
17    project_id INT AUTO_INCREMENT,
18    project_name VARCHAR(100) NOT NULL,
19    responsible_person_id INT NOT NULL,
20    PRIMARY KEY ( project_id )
21 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Special notes about the code:
AUTO_INCREMENT: to make the integer increment by one automatically. Usually applied to primary keys.
NOT NULL: to force the data field not to be null.
PRIMARY KEY: to set the data field to be the primary key for indexing data rows.
ENGINE: to set the database engine.
CHARSET: to set character set for this table.

We can write a complete definition of the tables upon creation. However, we are also able to alter a table after it is created. With the **ALTER TABLE** command, we are able to add a new column, drop an existing column, change a column's name and its data type, change the table name, add keys and so on. See Code 6.4.

Code 6.4: Alter the Table

```
1 ALTER TABLE staff ADD birthday DATE [AFTER address]; -- to ↩
    ↪ add a column birthday [after address]
2 ALTER TABLE staff DROP birthday; -- to drop the column ↩
    ↪ birthday
3 ALTER TABLE staff CHANGE name staff_name VARCHAR(40) NOT NULL↩
    ↪ ; -- to change the column name and its data type
4 ALTER TABLE project RENAME TO asic_project; -- to rename the ↩
    ↪ table project
```

In many cases we might want to add some constraints to tables by adding keys to them. The **UNIQUE** key constraint ensure values of a certain table column are unique. Inserting a value which already exists in the table will fail. We can also add a **UNIQUE** key to multiple columns. In this case, the composition of values corresponding to the columns must be unique. A **PRIMARY** key is similar to a **UNIQUE** key which is **NOT NULL**. Each table can have at most one **PRIMARY** key. It is a good practice to always define an ID column for each table and add a **PRIMARY** key to it. A **Foreign** key is a column in table 1 whose values match the **PRIMARY** key in table 2. Values of this column in table 1 must exist in the reference column in table 2.

We still take Code 6.3 for example. In this example, staff names must not duplicate. We can then add a **UNIQUE** key constraint to the column **name** of the table **staff**. Similarly, we might also want to add a **UNIQUE** key to the column **project_name** of the table **project**. When we create a new project, we have to specify the responsible person by his or her staff ID. This ID must already exist in the **staff** table.

Code 6.5: Add Key Constraints to Tables

```
1 ALTER TABLE staff ADD UNIQUE (name);
2 ALTER TABLE project ADD UNIQUE (project_name);
3 -- ALTER TABLE staff ADD PRIMARY KEY (staff_id);
4 -- ALTER TABLE project ADD PRIMARY KEY (project_id);
5 ALTER TABLE project ADD FOREIGN KEY (responsible_person_id) ↩
    ↪ REFERENCES staff (staff_id);
```

Thus, we can add a **Foreign** key to the column **responsible_person_id** of the table **project** referencing the column **staff_id** of the table **staff**. The **PRIMARY** keys have

been created with the tables. These keys can also be added afterwards using the **ALTER TABLE** command. See Code 6.5.

2. Data Operations

Data operations allow users to query data, insert new data entries to the database or update existing data entries. To insert a new data entry, we specify values for each data column and insert the data into a certain table using the **INSERT** command. To update a data entry, we have to specify the data columns to update and their new values. We usually have to specify which data entries to update using a **WHERE** clause. Similarly, when we delete data entries, we also need to specify a **WHERE** clause otherwise all data entries in that table will be deleted. In practice, every table usually has a ID column, which is an **AUTO_INCREMENT PRIMARY** key. If we know the value, the **WHERE** clause can be **WHERE id_column=1**. See Code 6.6.

Code 6.6: Insert, Update and Delete Data Entries

```
1 INSERT INTO staff ( staff_id, name, phone_number, address ) ↩
    ↪ VALUES ( 1, "Lei", "NaN", "Sichuan" );
2 INSERT INTO staff ( staff_id, name, phone_number, address ) ↩
    ↪ VALUES ( 2, "Bastl", "NaN", "Munich" );
3 INSERT INTO project ( project_id, project_name, ↩
    ↪ responsible_person_id ) VALUES ( 1, "IASRegisterManager"↩
    ↪ , 1 ); -- responsible person is Lei
4 INSERT INTO project ( project_id, project_name, ↩
    ↪ responsible_person_id ) VALUES ( 2, "Best␣Project", 2 );↩
    ↪  -- responsible person is Bastl
5 INSERT INTO project ( project_id, project_name, ↩
    ↪ responsible_person_id ) VALUES ( 3, "Bad␣Project", 1 );
6 UPDATE staff SET address="Aachen", phone_number="Unknown" ↩
    ↪ WHERE staff_id=1; -- Lei's address and phone_number ↩
    ↪ change
7 DELETE FROM project WHERE project_id=3; -- "Bad Project" is ↩
    ↪ deleted
```

To query data we use the **SELECT** command. We have to specify the data columns to select, or just use * as a shortcut to all columns. Like the **UPDATE** and **DELETE** command we might want to specify a **WHERE** clause unless we want to query all data entries in that table. Besides data columns, we can also **SELECT** a function. The data

value would be the result of that function. In many cases we might want to select data from multiple related tables. In this case, the **INNER JOIN** command might help. See Code 6.7.

Code 6.7: Data Query

```
1 SELECT address FROM staff WHERE name="Bastl"; -- result: ↩
    ↪ Munich
2 SELECT name, address FROM staff WHERE staff_id=1; -- result: ↩
    ↪ Lei, Aachen
3 SELECT COUNT(*) FROM staff; -- result: 2
4 SELECT staff.name, project.project_name FROM staff INNER JOIN↩
    ↪  project WHERE staff.staff_id = project.↩
    ↪ responsible_person_id;
5 -- result:
6 -- Lei, IASRegisterManager
7 -- Bastl, Best Project
```

6.2.2 Database Connector and Database Handler

MySQL uses the client/server architecture. Although we are able to connect to the server either by GUI tools such as MySQL Workbench or using the command line, to build a database application we have to use the MySQL Connector [14]. MySQL Connector is not a standard component of the MySQL package. It can be downloaded from MySQL website and installed easily.

To use the connector, we need to include the header files in our code and initialize the database connection. First, we should create a driver instance, and connect it to the database server given the host name, username and password. Then, we will create a statement instance. To execute a query statement, call the **executeQuery()** function, and parse the result set. To execute any statement that writes the database, call the **executeUpdate()** function.

We realized that the raw APIs offered by the MySQL Connector is not so friendly and reusable. One problem is that we have to write a complete SQL statement for each query or update. This is not really necessary, because the statement is highly structured. We

can write functions that accept only keywords, such as table name, column name, and generate complete statements. In this way, the API would be much more friendly to users. Another problem is that we have to manually parse the results for each query. Also, we can write a function to do this. Besides these, from an object-oriented programming point of view, it would be nice to wrap all these stuffs, the driver, the connection, statement and so on, in a class. So we designed a database handler class as in Code 6.8 and 6.9. In the code we omitted the database operation functions such as creating a table, querying data and so on.

Code 6.8: Header of the Database Handler

```
1  // database_handler.h
2  class DataBaseHandler
3  {
4  public:
5      static bool initialize(const QString& hostname, const ↩
           ↪ QString& database, const QString& username, const ↩
           ↪ QString& password);
6      static bool use_database(const QString &database);
7      static void close();
8      static void commit();
9      static void rollback();
10
11     /*
12      We define database operation functions here.
13     */
14
15     static bool get_next_auto_increment_id(const QString& ↩
           ↪ tablename,
16                                            const QString& id_field,
17                                            QString& id);
18     static QString get_error_message();
19
20 private:
21     static bool execute(const QString &statement);
22     static bool execute_query(const QString &statement);
23
24     static QString error_message_;
25     static QString database_;
26     static sql::Driver* driver_;
27     static std::unique_ptr< sql::Connection > con_;
```

```
28      static std::unique_ptr< sql::Statement > stmt_;
29      static std::unique_ptr< sql::ResultSet > res_;
30  };
```

Code 6.9: Source Code of the Database Handler

```
1   // database_handler.cpp
2   bool DataBaseHandler::initialize(const QString &hostname, ↩
       ↪ const QString &database, const QString& username, const ↩
       ↪ QString& password)
3   {
4       try {
5           con_.reset(driver_->connect(hostname.toUtf8().↩
               ↪ constData(),
6               username.toUtf8().constData(),
7               password.toUtf8().constData()));
8           con_->setAutoCommit(false);
9           if (database != "")
10          {
11              con_->setSchema(database.toUtf8().constData());
12              database_ = database;
13          }
14          stmt_.reset(con_->createStatement());
15          return true;
16      } catch (sql::SQLException &e) {
17          error_message_ = e.what();
18          return false;
19      }
20  }
21
22  bool DataBaseHandler::execute(const QString &statement)
23  {
24      try {
25          stmt_->executeUpdate(statement.toUtf8().constData());
26          return true;
27      } catch (sql::SQLException &e) {
28          error_message_ = e.what();
29        return false;
30      }
31  }
32
33  bool DataBaseHandler::execute_query(const QString &statement)
34  {
```

```
35      try {
36          res_.reset(stmt_->executeQuery(statement.toUtf8().↩
               ↪ constData()));
37          return true;
38      } catch (sql::SQLException &e) {
39          error_message_ = e.what();
40        return false;
41      }
42  }
```

This initialization of the database connection is wrapped in the **initialize()** function. After initialization, we can do database operations. Every database operation function parses its parameters and generate a SQL statement string, and either calls the **execute()** or the **execute_query()** function, depending on whether it is an update or a query operation. The return type of the database operation functions should always be bool so that we know whether they are successful. In practice we want to execute a series of SQL operations as a work unit which we call a transaction. One property of the transaction is atomicity. It requires that all SQL operations must be successful, or we have to move back to the point before this transaction. After each transaction we have to determine whether the conjunction of returned values are true, and **commit()** the transaction or **rollback()**. If a transaction fails, we can get the error message using the **get_error_message()** function. See Code 6.10.

Code 6.10: Example of the Database Transaction

```
1  // database transaction example
2  bool success = true;
3  success = success && DataBaseHandler::some_operation();
4  success = success && DataBaseHandler::another_operation();
5  if (success) DataBaseHandler::commit();
6  else
7  {
8      DataBaseHandler::rollback();
9      QString error_message = DataBaseHandler::↩
          ↪ get_error_message();
10     // do something
11 }
```

It should be noted that all member variables and functions inside the **DataBaseHandler**

class are *static*. This is because the **DataBaseHandler** is very frequently used. We found it would be very inefficient if we create a **DataBaseHandler** instance, initialize the database connector every time we want to access the database. To make everything static, we only have to initialize the database connector and connect to the database server once, and the same connector instance is used every time and everywhere. We must initialize the database connector upon initialization of the software.

6.3 Database Design

Based on an existing draft, we designed the database structure consisting of a number of tables. The tables, data fields and types of each table and relationships between tables have be been illustrated in the Figure 6.2.

It is important that we do not hardcode e.g. project roles and register types in the software. Instead, they are defined in the tables starting with **def** and this provides us much flexibility. For example, we can add a new project role by simply adding a new row to the **def_project_role** table without modifying the software itself.

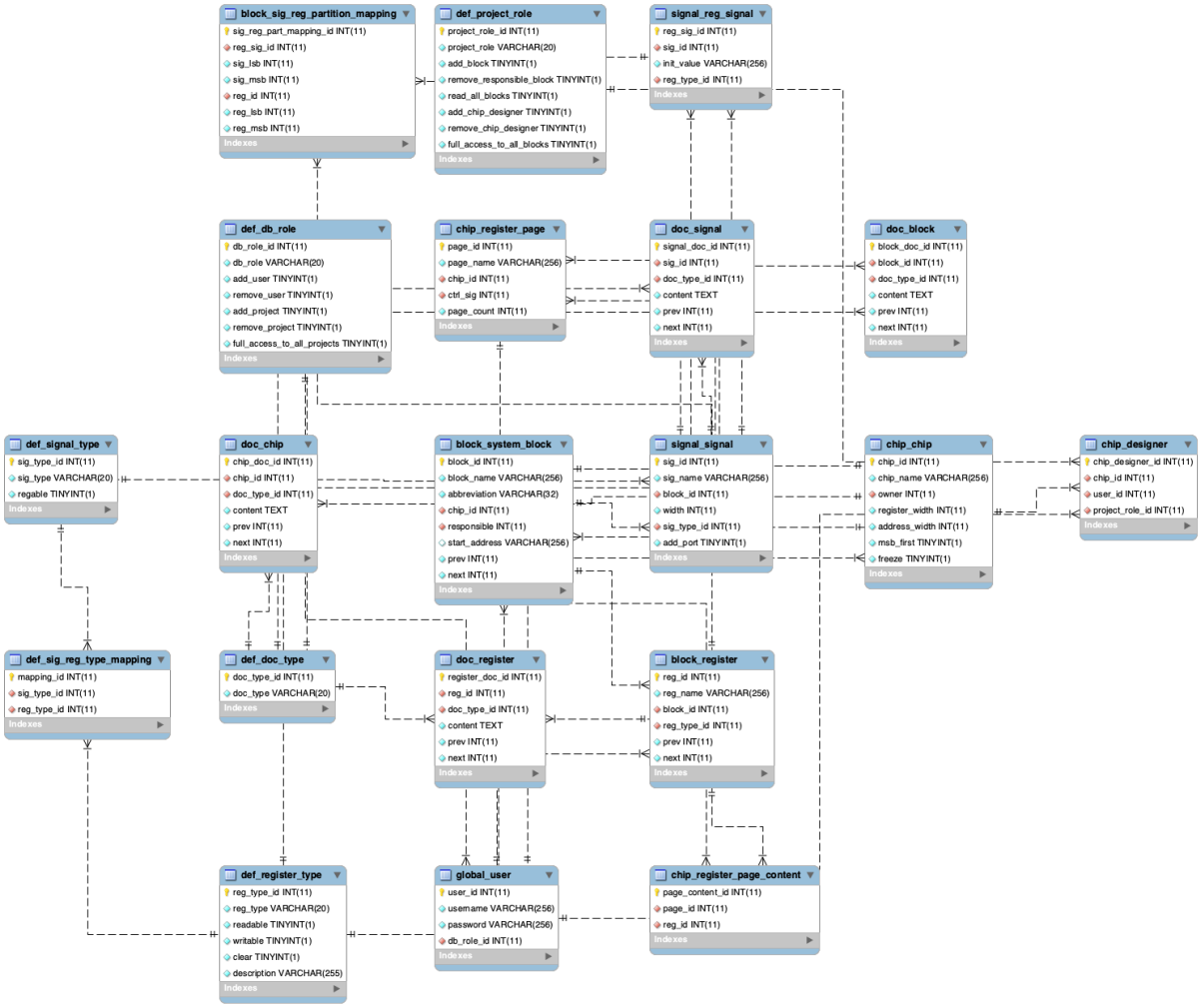


Figure 6.2: Database Structure

7 Module Design and Implementation

So far we have designed the software at the system level and learned background knowledge for implementation. In this chapter, we will design and implement each class.

7.1 Main Window

According to the requirements, we designed the main window as shown in Figure 7.1. The main window is not a unity as it seems to be. In fact, four classes are involved here: **RegisterManager**, which is the framework of the main window, **ChipNavigator**, **ChipEditorView** and **DocumentEditorView**. Their user interfaces are independently designed and compose the whole main window.

We have described in the Chapter 5 how these components work together. When the current item of the **ChipNavigator** changes, the **ChipEditorView** and the **DocumentEditorView** have to respond. To make this happen, we defined a slot function in the **ChipNavigator** to deal with the predefined **currentItemChanged()** signal of the tree widget. According to the level of the current tree widget item, we know whether it is the chip, a system block, a register, or a signal. Then, we defined signal functions for each level and emit them accordingly. The logic can be described by Code 7.1.

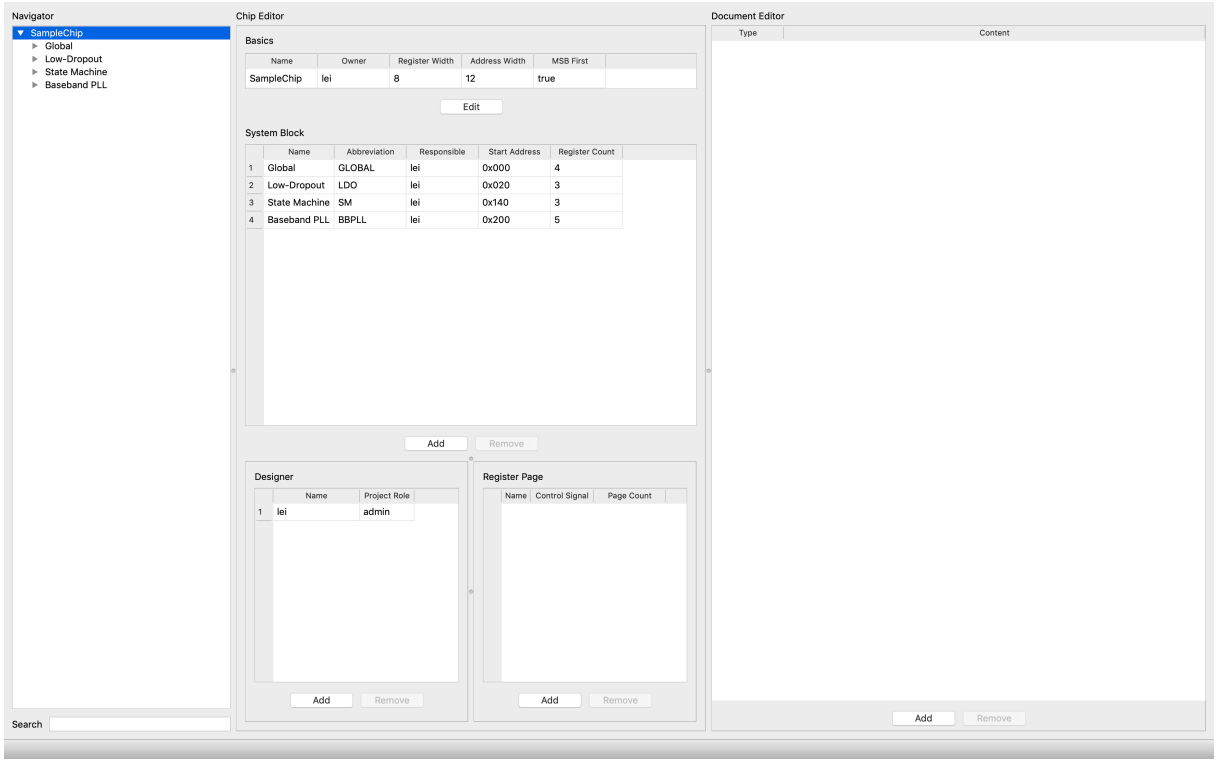


Figure 7.1: Main Window

Code 7.1: Logic of Chip Navigation: Emission of Signals

```
1  void  on_treeWidgetBlock_currentItemChanged(QTreeWidgetItem *↩
       ↪ current, QTreeWidgetItem *previous)
2  {
3      if (level(current) == LEVEL::CHIP)
4          emit(chip_clicked());
5      if (level(current) == LEVEL::BLOCK)
6          emit(block_clicked(get_block_id(current)));
7      if (level(current) == LEVEL::REGISTER)
8          emit(register_clicked(get_block_id(current), ↩
             ↪ get_register_id(current)));
9      if (level(current) == LEVEL::SIGNAL)
10         emit(signal_clicked(get_block_id(current), ↩
             ↪ get_signal_id(current)));
11 }
```

To take care of the signal, we defined slot functions in the **RegisterManager** class (see

Code 7.2) and connect them to corresponding signals. In the slot functions, we first update the current block, register and signal ID. Then, we reset the document level. We pass the IDs to the **ChipEditorView** and **DocumentEditorView**, then let them display documents and chip information respectively. See Code 7.3.

Code 7.2: Logic of Chip Navigation: Declarations of Slot Functions

```
// register_manager.h
void on_chipNavigator_chip_clicked();
void on_chipNavigator_block_clicked(QString block_id);
void on_chipNavigator_register_clicked(QString block_id,
        QString reg_id);
void on_chipNavigator_signal_clicked(QString block_id,
        QString sig_id);
```

Code 7.3: Logic of Chip Navigation: Definitions of Slot Functions

```
// register_manager.cpp
void RegisterManager::on_chipNavigator_register_clicked(↩
   ↪ QString block_id, QString reg_id)
{
     ui->docEditorView->set_doc_level(LEVEL::REGISTER);
     ui->docEditorView->set_block_id(block_id);
     ui->docEditorView->set_register_id(reg_id);
     if (ui->frameDoc->isVisible() && ui->actionDocEditor->↩
        ↪ isChecked())
         ui->docEditorView->display_documents();

     ui->chipEditorView->set_block_id(block_id);
     if (ui->frameChipEditor->isVisible())
         ui->chipEditorView->display_system_level_info(reg_id)↩
            ↪ ;
     current_block_id_ = block_id;
     current_reg_id_ = reg_id;
     current_sig_id_ = "";
}
```

We can edit the chip in the **ChipEditorView** by editing the chip name, adding, editing or removing system blocks, registers, signals, signal-register mappings and so on. In these cases, the chip navigator have to be updated. Similarly, we define signals in the **ChipEditorView** as in Code 7.4, and corresponding slot functions in the **RegisterManager**.

When the chip is edited, a certain signal is emitted and information is passed to the corresponding slot function in the **RegisterManager**. In the slot function, a certain function of the chip navigator is called to update the tree widget.

Code 7.4: Signals for Updating the Chip Navigator

```
// chip_editor_view.h
void chip_basics_edited(QString chip_name,
                        QString chip_owner,
                        QString chip_owner_id,
                        int register_width,
                        int address_width,
                        bool msb_first);
void block_added(QString block_id,
                        QString block_name,
                        QString block_abbr,
                        QString responsible);
void block_removed(int row);
void block_modified(int row,
                        QString block_name,
                        QString block_abbr,
                        QString responsible);
void block_order_exchanged(int from, int to);
void to_refresh_block();
```

The boundaries between the **ChipNavigator**, **ChipEditorView** and **DocumentEditorView** are adjustable. Users can also switch on or off the **ChipEditorView** or the **DocumentEditorView**.

We designed a menu bar for the main window. There are four menus as follows, each containing a number of menu entries, or actions.

- **User Menu**
  User Management: to open a **UserManagementDialog** in which users can add or remove users. It is only accessible to database admins.
  Change Password: to open a **ChangePasswordDialog** for users to change their password.
  Log Out: to log out of the current user account and to log in with another one.

- **Chip**
  New Chip: to open an **EditChipDialog** to create a new chip.
  New Chip From: to create a new chip from an existing one.
  Open Chip: to open an **OpenChipDialog** and open the selected chip.
  Close Chip: to close the current chip and clear everything in the **ChipNavigator**, **ChipEditorView** and **DocumentEditorView**.
  Naming: to open a **NamingTemplateDialog** to display the namings for registers or signals.
  Resources Base Dir: to open a **ResourcesBaseDirDialog** in which users can specify the base directory where resources such as figures are located.
  Freeze/Unfreeze: to freeze or unfreeze the current chip.
  Chip Management: to open an **OpenChipDialog** and configure it to the management mode. Users can add or remove chips. It is only accessible to database admins.

- **Export**
  SPI Source Code: to open an **SPIGenerationDialog** and generate the SPI interface.
  Document: to open an **DocumentGenerationDialog** and generate the documentation.

- **View**
  Chip Editor: to enable or disable the **ChipEditorView**.
  Document Editor: to enable the **Document Editor** page of the **DocumentEditorView**.
  Document Preview: to enable the **Document Preview** page of the **DocumentEditorView**. The **Document Editor** and **Document Preview** actions are exclusively checkable.

7.1.1 ChipNavigator

We have described the logic of the main window and its interaction with the **ChipNavigator**, the **ChipEditorView** and the **DocumentEditorView**. The **ChipNavigator**

is quite simple. According to requirements, the structure of the chip is represented with a tree widget of four levels: CHIP-BLOCK-REGISTER-SIGNAL. The root of the tree is the chip level, with its children being system blocks. Under each system block are registers. The children of each register are those signals mapped to it.

We implemented a search mechanism which might be very beneficial especially when the chip is complex. The users can simply type in the pattern to search, and the tree widget will highlight those components matching that pattern and make others invisible. The search process is written in a single **search()** function. Whenever the input pattern changes, the **search()** function will be executed in response to the **textChanged()** signal from the tree widget. Figure 7.2 shows an example of the search process.

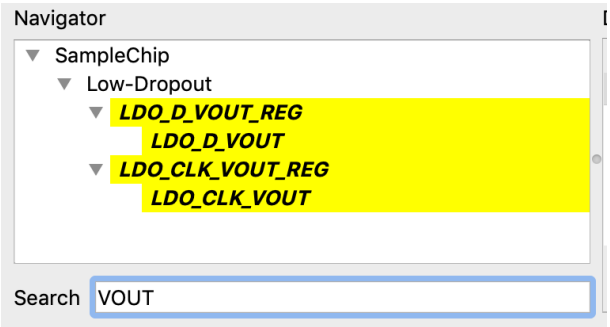


Figure 7.2: Chip Navigator: Search Example

7.1.2 Chip Editor View

The **ChipEditorView** is one of the three major components of the main window. It displays information about the chip and provides entrance to dialogs for editing the chip. According to the requirements, we designed the **ChipEditorView** as in Figure 7.3 and 7.4.

Whether the chip level or the system level page is displayed depends on the current item of the navigator. If it is the root of the tree widget, then the **RegisterManager** class will call the **display_chip_level_info()** function of the **ChipEditorView**, otherwise **display_system_level_info(register_id, signal_id)** is called. If the current item is a register, then its **register_id** will be passed to the function and **signal_id** will be null.

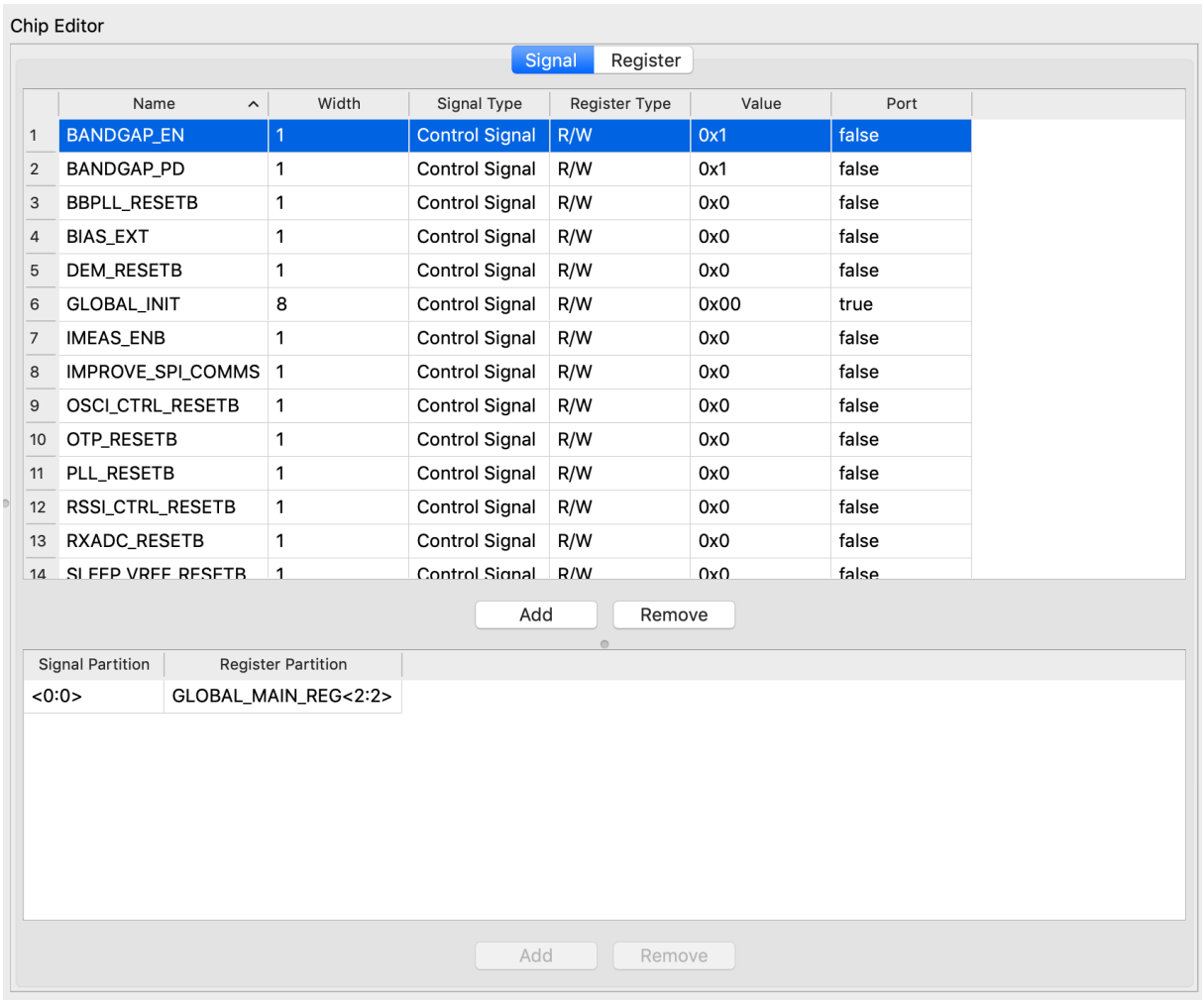


Figure 7.3: Chip Editor: Signal Tab

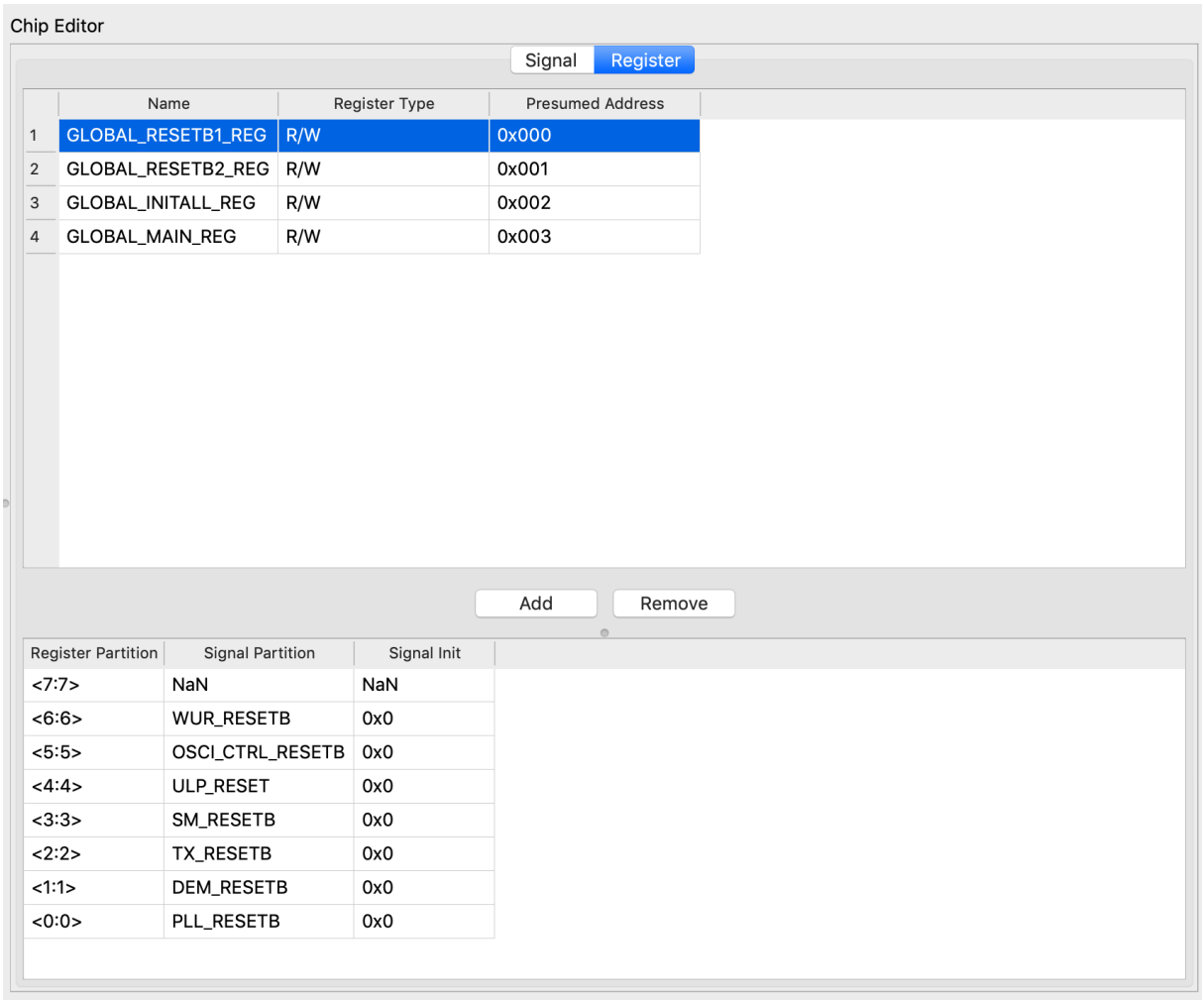


Figure 7.4: Chip Editor: Register Tab

Likewise, if the item is a signal, then **register_id** will be null. Depending on whether **register_id** or **signal_id** is null, the **ChipEditorView** will display the register page or the signal page, and set the selected register or signal in the register or the signal table as active. If the current item is a block, both **register_id** and **signal_id** are null, and the **ChipEditorView** simply displays the current page. This process can be described with the following simplified Code 7.5.

Code 7.5: Logic of Displaying System Level Information

```
1  // chip_editor_view.cpp
2  void ChipEditorView::display_system_level_info(const QString&↩
       ↪  reg_id, const QString& sig_id)
3  {
4      // set to system level page
5      ui->stackedWidgetChipEditor->setCurrentIndex(1);
6
7      // set to register or signal page
8      if (sig_id != "") ui->tabWidget->setCurrentIndex(0);
9      else if (reg_id != "") ui->tabWidget->setCurrentIndex(1);
10
11     // display either signals or registers
12     if (ui->tabWidget->currentIndex() == 0) display_signals()↩
           ↪ ;
13     else display_registers();
14
15     // set current signal as active
16     if (sig_id != "")
17     {
18         for (int row = 0; row < ui->tableSignal->rowCount(); ↩
               ↪ row++)
19             if (sig_id == ui->tableSignal->item(row, 0)->text↩
                   ↪ ())
20             {
21                 ui->tableSignal->setCurrentCell(row, 0);
22                 break;
23             }
24     }
25     // set current register as active
26     else if (reg_id != "")
27     {
28         for (int row = 0; row < ui->tableRegister->rowCount()↩
               ↪ ; row++)
```

```
29                 if (reg_id == ui->tableRegister->item(row, 0)->↩
                       ↪ text())
30                 {
31                     ui->tableRegister->setCurrentCell(row, 0);
32                     break;
33                 }
34         }
35     }
```

To add a system block, register, signal or a signal-register mapping we provide a push button under each table. After a click on the button a dialog of the corresponding class will be created and open. To remove an existing item we can click on the **Remove** buttons leading to execution of a corresponding slot function. The function will first remove the system block, register etc. in the database, then remove it from the table. To edit an existing system block, register or signal, we can double click the table entry. The **cellDoubleClicked()** signal will then be emitted. We write corresponding slot functions to help us edit the items. The logic is actually very similar to adding a new item. A corresponding dialog will open, but it will be initialized with data related to that item. The dialogs will be discussed in the Section 7.2.

For users' convenience, we designed a right-click context menu containing an **Edit**, **Remove** and **Add** action. Users do not have to click on the push buttons below each table. The menu also contains a **Refresh** action, which allows for reloading data from the database and refreshing the table. Once we right-click on the tables, a **customContextMenuRequested()** signal will be triggered. A slot function is then called to display the context menu in response to the signal.

7.1.3 Document Editor View

The **DocumentEditorView** shown in Figure 7.5 is another central component of the software. It displays document items and provides functionality for editing them. According to the requirements, it should contain a table widget showing all document items of the current item on the chip navigator. At the bottom of the **DocumentEditorView** there is an editor "dialog" in which we can add or edit a document. It is an instance of the

EditDocumentDialog class, which is not a real dialog though, but a subclass of the generic **QWidget**. This is because the **QDialog** cannot be a child widget of another Qt widget. The dialog can be switched on or off. Like the **ChipEditorView**, we can add or edit document items by clicking on the **Add** button below or double clicking a table entry. The editing dialog will then open. If we have finished editing and want to save the document, we click on the **Ok** button. The document will be saved to the database and the table above will be updated. Then, the editing area will be switched off. If we want to abandon the document we are editing, simply click on the **Cancel** button.

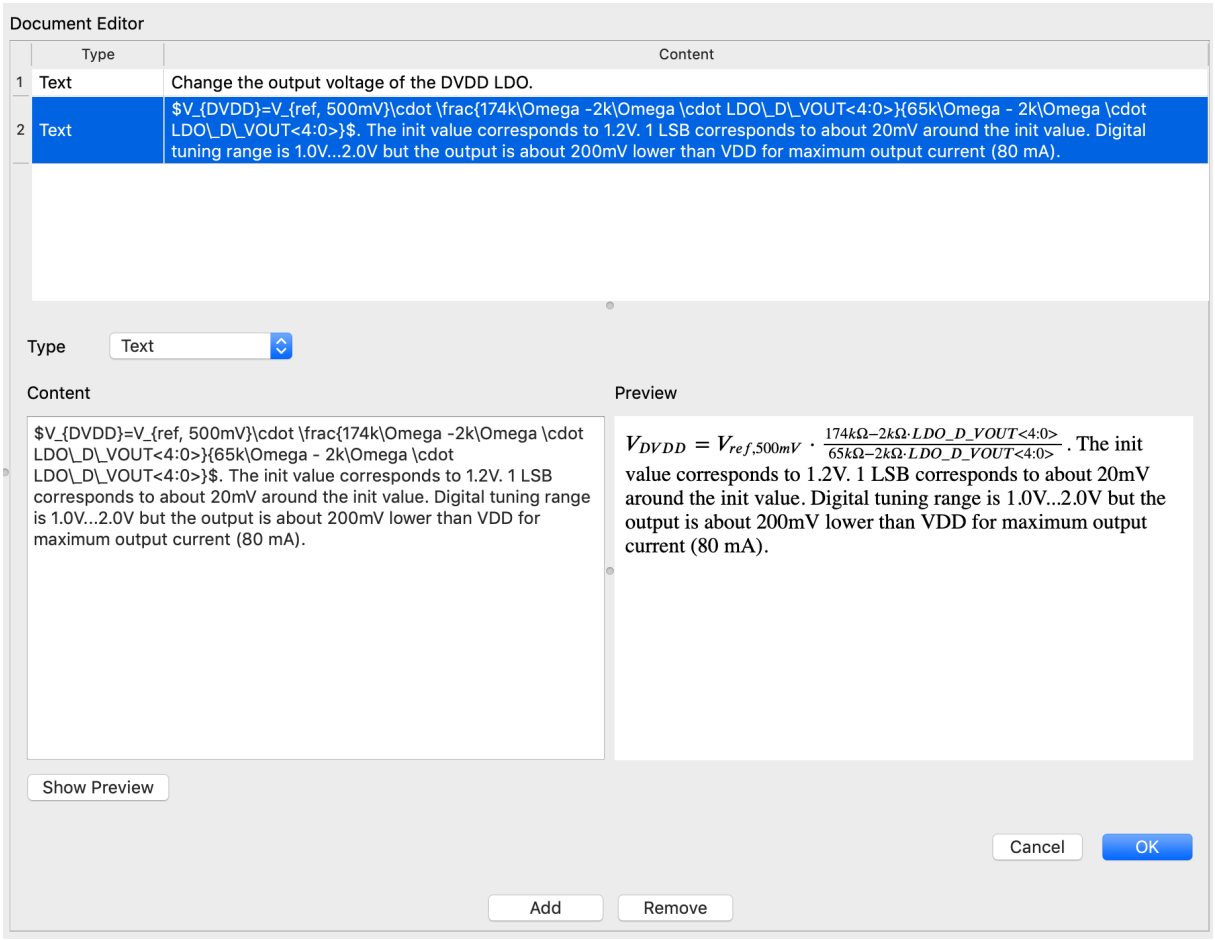


Figure 7.5: Document Editor View

Apart from the **Add**, **Remove** and **Edit** functions, we provided a copy and paste mechanism, which might be very useful for documentation editing. To achieve this, we maintain

a member variable **copied_** of type **QString**. When we copy a document, the corresponding document type ID, document type, and the content to copy will be concatenated by a special delimiter and then be stored in this **copied_** variable. When we paste the copied content, the **copied_** variable will be parsed to fetch the document type ID, document type, and the content to paste. With this information, we can create a new entry in the document table and store this document to the database.

7.2 Chip and Document Editor Dialogs

To edit chip components and document items, we have to design different dialogs encapsulating both UI and business logics. These dialogs are named after **EditSomethingDialog**, such as **EditChipDialog** and **EditSignalDialog**. They are defined following the framework in Code 7.6.

Code 7.6: Framework for Chip and Document Editor Dialog Classes

```
1  // edit_something_dialog.h
2  class EditSomethingDialog : public QDialog
3  {
4      Q_OBJECT
5
6  public:
7      explicit EditSomethingDialog(QWidget *parent = nullptr);
8      explicit EditSomethingDialog(QString something_id, bool ↩
           ↪ enabled, QWidget *parent = nullptr);
9      // other construction functions
10     ~EditSomethingDialog();
11     QString get_something_id();
12     // Other public functions
13
14     bool add_something();
15     bool edit_something();
16
17 private:
18     bool setup_ui();
19     void accept(); // override QDialog::accept()
20     bool sanity_check();
21     /*
```

```
22          Things to check
23      */
24      // Other private functions
25
26      Ui::EditSomethingDialog *ui;
27      const bool enabled_;
28      const DIALOG_MODE mode_;
29      QString something_id_;
30      // Other private member variables
31  };
```

The class has at least two construction functions, one for adding, the other for editing something. The **DIALOG_MODE mode_** will correspondingly set to **ADD** or **EDIT**. Upon construction the dialog first sets up the UI by calling the **setup_ui()** function. If the dialog is constructed in the **EDIT** mode, it will pass the ID to the dialog and fill in the existing data into the widgets. Also, there is a parameter called **enabled**. If it is false, then the UI components will be disabled so users cannot put in anything to the dialog. Otherwise, users give their inputs to the dialog and click on the **Ok** button, and the **accept()** function will be executed.

By default, if the **EditSomethingDialog** is executed as modal, i.e. by calling the **exec()** function, the **QDialog::accept()** function will close the dialog and the **exec()** function will return **QDialog::Accepted**. However, in our cases things are different. If the dialog is not enabled, we just want it to close and return **QDialog::Rejected**. This is equivalent to clicking on the **Cancel** button. Otherwise, we do sanity checks, which is one of the system requirements. The purpose of the sanity checks is to determine whether the input data is valid. If all checks are successful, then **accept()**, else do nothing to the dialog. The dialog will stay open for users to modify their inputs. The overridden **accept()** function and the **sanity_check()** function are defined in Code 7.7.

Code 7.7: Acceptance Logic of Editor Dialogs

```
1  // edit_something_dialog.cpp
2  void EditSomethingDialog::accept()
3  {
4      if (!enabled_) return QDialog::reject();
5      if (sanity_check()) return QDialog::accept();
6  }
```

```
7
8  void EditSomethingDialog::sanity_check()
9  {
10     return check1() && check2() && check3();
11 }
```

We can determine what we want to check. The check functions basically follow the same pattern as in Code 7.8.

Code 7.8: Example of Sanity Check Functions

```
1  // edit_something_dialog.cpp
2  bool EditSomethingDialog::check_something_name()
3  {
4      QString name = get_something_name();
5      if (name == "")
6      {
7          QMessageBox::warning(this, "Add␣Something", "↩
               ↪ Something␣name␣must␣not␣be␣empty!");
8          return false;
9      }
10     if (mode_ == DIALOG_MODE::EDIT && name == original_name_)↩
           ↪  return true;
11     if (exists_in_database(name))
12     {
13         QMessageBox::warning(this, "Add␣Something", "↩
               ↪ Something" + name + "already␣exists!")
14         return false;
15     }
16     return true;
17 }
```

They check something, prompt error messages and return false at once if a check fails, or return true if all checks passes. In this example, we want to verify if the name is valid. The name must not be empty or duplicated. If it is empty, a warning message will be prompted and the function will return false. Otherwise, if the dialog is in the **ADD** mode, we have to fetch names from the database and check if the input name already exists. If so, users are warned and the function returns false. If everything is fine, then the function returns true. It is a little different in the **EDIT** mode. In that case, the original name

actually exists in the database. If the current updated name equals the original name, the function just returns true, otherwise we check it as in the **ADD** mode.

If the **exec()** function returns true, we then execute either the **add_something()** or **edit_something()** function. Then, we get what data we need from the dialog. The logic is shown in Code 7.9.

Code 7.9: Logic of Adding or Editing Something

```
// somewhere_using_this_dialog.cpp
// add something
EditSomethingDialog add_somthing(this);
If (add_somthing.exec() == QDialog::Accepted && add_somthing.↩
   ↪ add_something())
{
     QString something_id_ = add_somthing.get_something_id();
     // Other data we want to get
     // Do something
}

// edit something existing
QString something_id; // got from somewhere
bool enabled; // got from somewhere
EditSomethingDialog edit_somthing(something_id, enabled, this↩
   ↪ );
If (edit_somthing.exec() == QDialog::Accepted && ↩
   ↪ edit_somthing.edit_something())
{
     // Other data we want to get
     // Do something
}
```

The most complex editor dialogs are the **EditSignalDialog** and **EditDocumentDialog**. The **EditSignalDialog** allows users to add or edit signals. Users have to put in the signal name, width i.e. the number of bits, signal type and whether to add a port for this signal. If the signal is writable, users have to put in its initial value. If the signal can be mapped to a register, or in other words, it is a register-signal, the dialog allows users to edit signal-register mappings. A signal can only be mapped to one type of registers.

In practice we found almost half of the signals are single-bit signals. In this case, it can

simply be mapped to a certain bit of a register as a whole. In other cases where the signal have more bits, intuitively we can map it to registers bit by bit. To map an 8-bit signal, for example, eight database operations are required. To reduce database operations and increase reliability, we proposed to map signals and registers by continuous bit blocks which we call partitions. We have to partition the signal and corresponding registers. Each signal partition is then mapped to a register partition with the same number of bits. This can be illustrated with Figure 7.6.

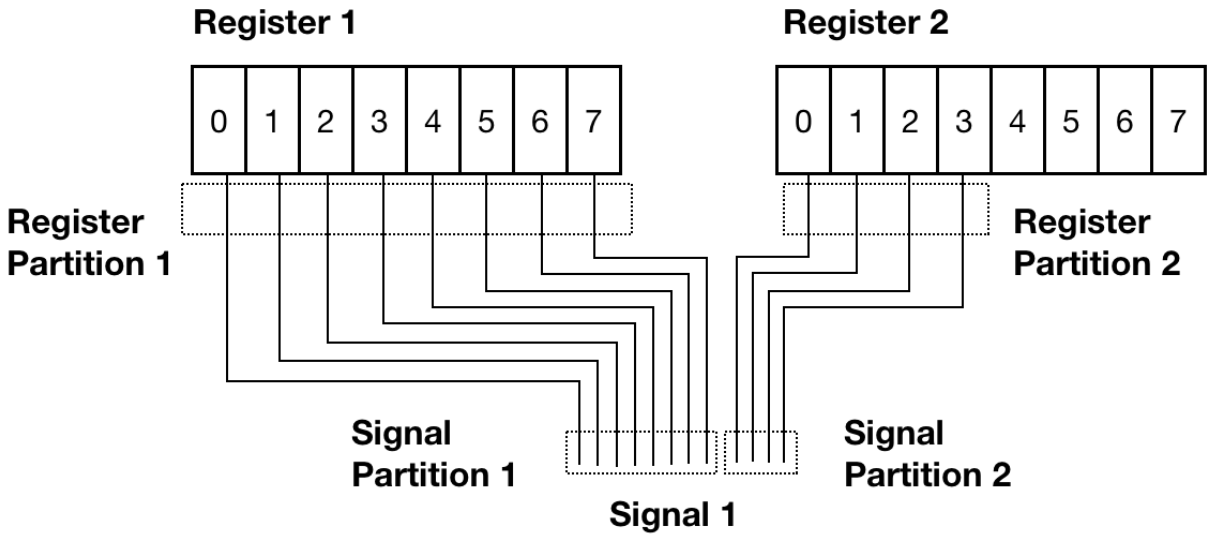


Figure 7.6: Mappings of Signal-Register Partitions

Therefore, we designed UI for two cases shown in Figure 7.7 and 7.8. In case of a single-bit signal, users just need to select a certain register and a certain bit of it to map the signal to. Users don't have to explicitly add a signal-register mapping to the partition list. In case of a multi-bit signal, however, users must determine the signal partition by giving its MSB and LSB, select a register, and a register partition. Then, add the signal-register partition to the candidate list. Of course, users can also remove signal-register mapping from the list. To make it more user friendly, we can add a new register by clicking on the button beside the register combo box.

Widgets in the **EditSignalDialog** have pretty strong interaction with each other. Whenever the signal width changes, the signal-register mappings might be invalid, thus we have to clear the candidate list. Also, we have to adjust the UI according to the signal width. If

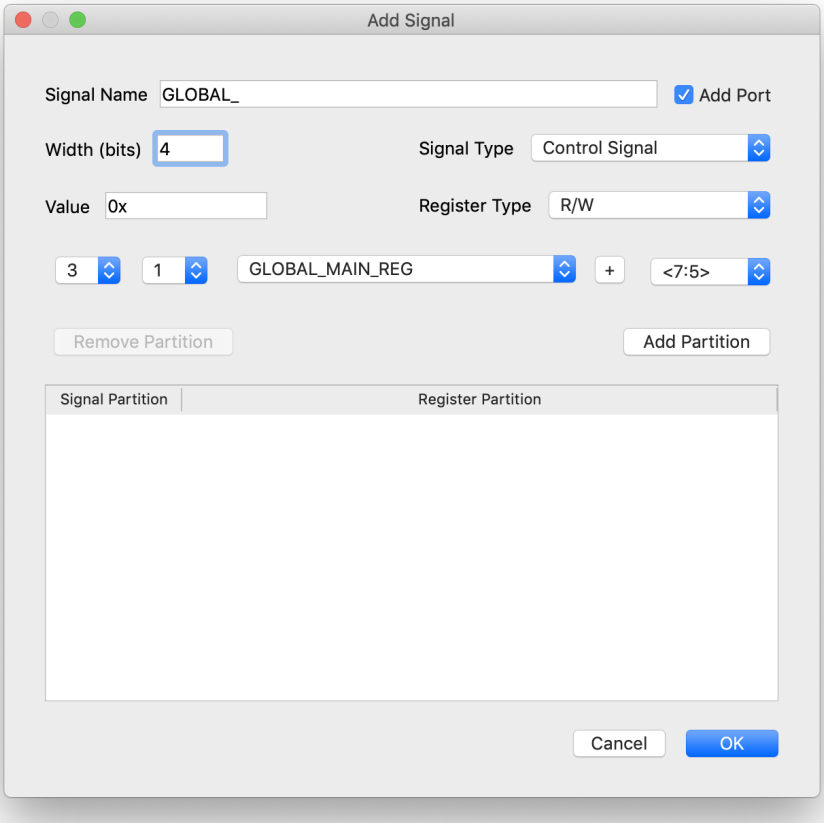


Figure 7.7: Editor UI for Multi-bit Signals

the signal type changes, we have to retrieve from the database the register types available for this signal type, and change the items in the register type combo box. If the current register type changes, we have to clear the signal-register mapping candidate list, and retrieve registers of such type from the database. Whenever the candidate list changes, we also have to find available signal partitions. It is crucial that a signal bit cannot be mapped to more than one register bit, and the other way around. The register partition combo box changes according to the signal LSB and MSB combo boxes. The signal/slot technique provided by Qt makes these changes much more tractable.

The **EditDocumentDialog** allows users to add document items to the current item in the **ChipNavigator**, or edit existing ones. It is named after **dialog** but it is actually not,

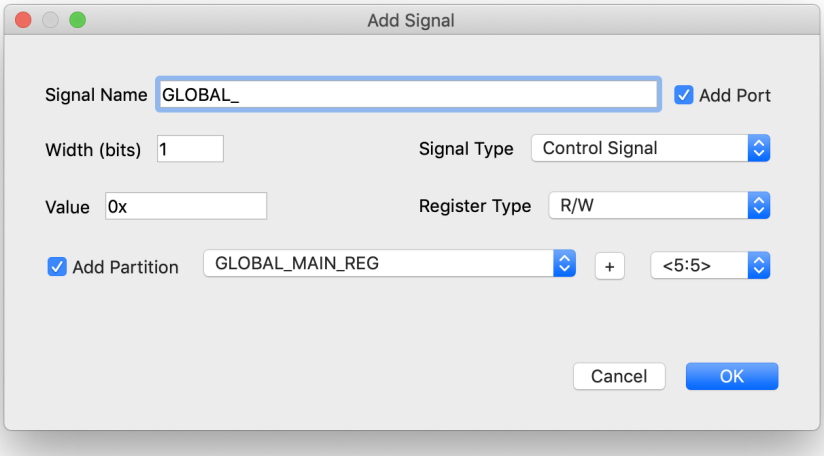


Figure 7.8: Editor UI for Single-bit Signalsl

because we want to incorporate it in the **DocumentEditorView**, and this is not possible for a dialog. Instead, it is a subclass of **QWidget**. However, we designed it following the the pattern of the editor dialogs.

We defined three document types, text, image and table. Thus, the **EditDocumentDialog** was designed so that it can take in all document types. To make this possible the dialog has a stacked widget containing three pages, each handling a specific document type. In the previous section the **DocumentEditorView** example in Figure 7.5 has already shown the page for text documents. The image page and the table page are shown in Figure 7.9 and 7.10 respectively.

The text page has a **QPlainTextEdit**, in which we can edit multiple lines of texts. In the image page, we can select an image in a **QFileDialog**, get its path and specify a caption. It is also important to specify how large the image is in terms of the page width. The number can vary between 0 and 1. In the table page, a **QLineEdit** holds the caption of the table. The **QTableWidget** allows users to edit the table contents conveniently. We can specify the number of rows and columns for the table.

We will store the document in the database. The question is how to deal with different

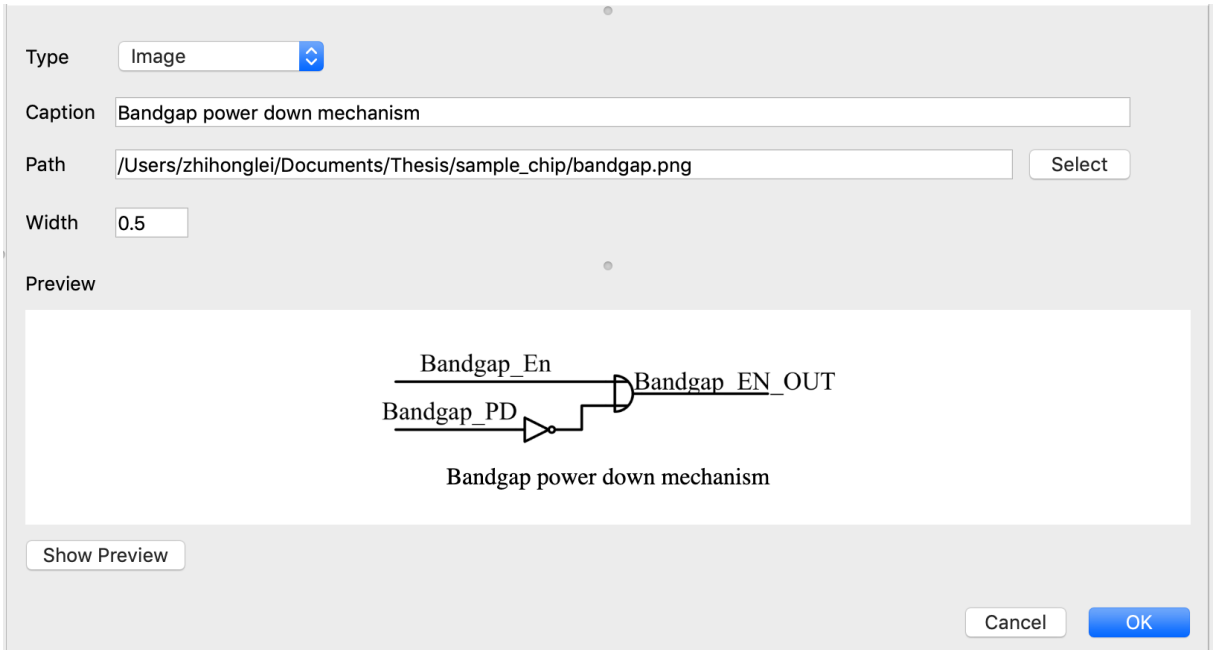


Figure 7.9: Image Document

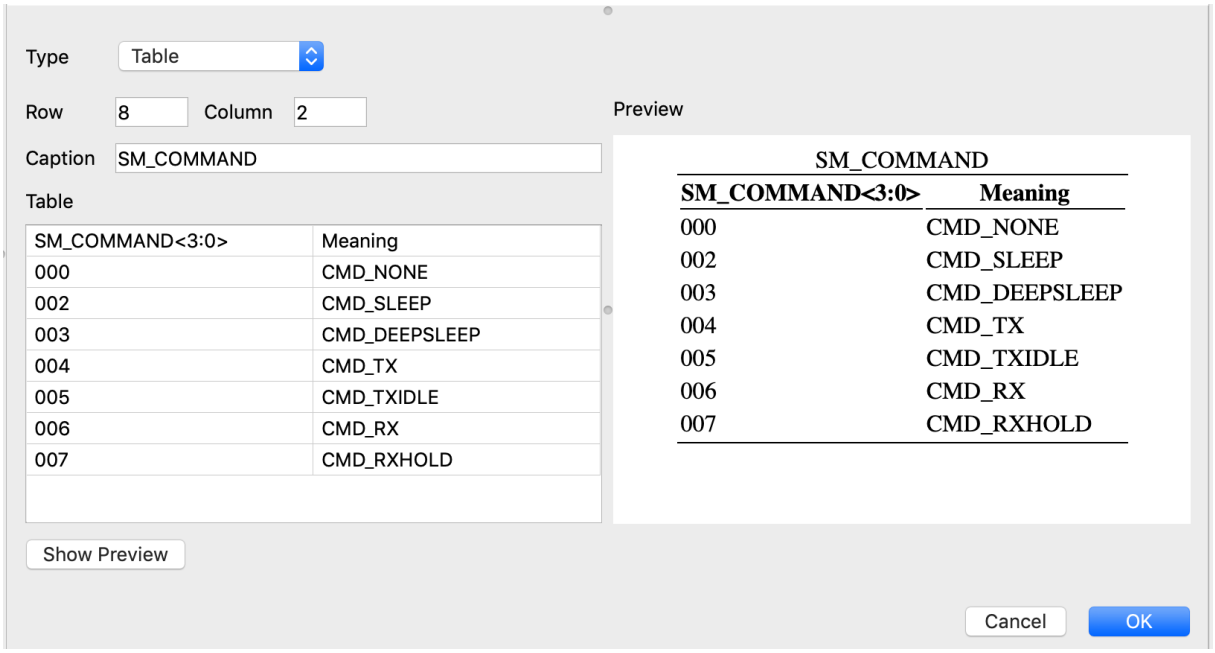


Figure 7.10: Table Document

types of documents. The text document is simple. We just store the text in the database. For image documents, our solution is to concatenate the caption, width, and the path with a special delimiter, and store the concatenated string in the database. Similarly, for the table documents, we concatenate the caption, row count, column count, and all cell contents. In this solution, of course the image caption, table caption and cell contents must not contain the delimiter. It would be a good idea to use a very rare string as the delimiter. We can easily restore the documents in a reverse way.

It is inevitable that we might have to include math equations in LaTeX format in the documents. In this case, of course we can type LaTeX code "blindly". It would be very nice if we have a preview of the LaTeX math equations. In this way, the input is more visual, and we can check if the code is correct. We also want to have a preview of the table and image. It is important that table cell contents can also contain LaTeX math equations.

To implement the preview, we might want to compile the LaTeX source code to generate a PDF document. Then, we must find a way to display the PDF document in our software. This is not a good idea. First, users have to install a LaTeX compiler on their machines and it must be accessible to the software. Second, Qt does not have a PDF render. We were able to find third-party renders, but they are either of low quality or hard to incorporate in our software.

However, we found MathJax [15], a JavaScript display engine for mathematics that works in web browsers. With MathJax included in the HTML web page, LaTeX source code can be simply inserted in the HTML code, and browsers can render it properly. We use MathJax with the template in Code 7.10.

Code 7.10: HTML Template for MathJax Render

```
1 <!-- HTML template -->
2 <html>
3 <head>
4     <script type="text/x-mathjax-config"> MathJax.Hub.Config↩
        ↪ ({
5         tex2jax: {inlineMath: [['$','$'], ['\\(', '\\)']]}});↩
            ↪ </script>
```

```
6       <script type="text/javascript" src="MATHJAX_ROOT/MathJax.↩
          ↪ js"></script>
7   </head>
8   <body>
9   HTML_CONTENT     <!-- write your HTML code here -->
10  </body>
11  </html>
```

Inspired by this, we can display text, image and table documents in a web browser. For text documents it is quite simple. We just replace **HTML_CONTENT** in the HTML template with the text in the text editor. For image and table documents, we created a template respectively, see Code 7.11 and 7.12. Using the image or table template, we generate image or table HTML source code, and replace **HTML_CONTENT** in the HTML template.

Code 7.11: HTML Template for Images

```
1   <!-- HTML image template -->
2   <style>
3       figure img {display: block; margin-left: auto; margin-↩
          ↪ right: auto;}
4       figure figcaption { text-align: center;}
5   </style>
6   <center>
7   <figure>
8      <img style='width: WIDTH; object-fit: contain' src="IMAGE"↩
         ↪  alt="CAPTION"/>
9       CAPTION
10  </figure>
11  </center>
```

Code 7.12: HTML Template for Tables

```
1   <!-- HTML table template -->
2   <style>
3       table {border-top: 1px solid black; border-bottom: 1px ↩
          ↪ solid black}
4       th {border-bottom: 1px solid black}
5       caption#tab {caption-side: CAPTION_POS}
6   </style>
7   <center>
```

```
8  <table>
9  TABLE
10 </table>
11 </center>
```

Fortunately, Qt provides a web engine [16] based on Chromium [17]. Thus, we were able to make a preview based on the web engine. The preview is intrinsically a web browser. To display documents of any types, we generate the HTML source code and fill it into the browser. In this way, the document editor can display the real time preview of the text users are editing.

Another effort we made is to implement an auto-completer. This is especially useful for editing texts. The completer we designed takes all system block names, system block abbreviations, register names and signal names as candidates. Besides these, the software allows users to extend the word list by simply add txt files containing candidate words to the **completion** directory under the root directory of the software. Figure 7.11 shows an example of auto-completion on the text document. We also implemented auto-complement for captions and table cell contents.

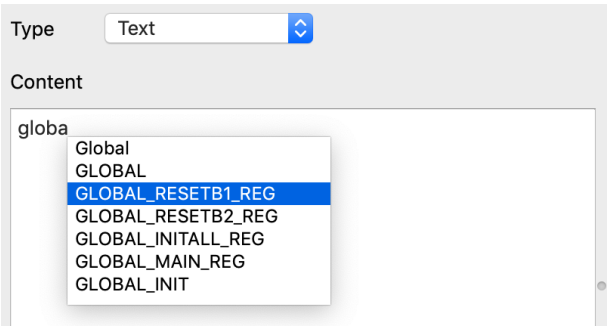


Figure 7.11: Document Completer

7.3 User Account, Authentication and Password

For data security reasons, the software is required to provide user access control and authentication. User access control has two levels, the software or database level and the project level. If a user account is created, the user can log in to the software with this

account. However, it does not necessarily mean he or she has access to a certain chip project. This is controlled by the project level permissions.

All users must have access to the database, thus, we need to establish database accounts for them. MySQL provides a very good user account management. We can grant different permissions for different users. However, the permissions we want to implement are more complex. For example, in a certain table, a user might have access to row A but not to row B. This is not possible with MySQL. Thus, we have to manage permissions using our software. Against this background, we distinguished the *software* account and *database* account. We store *software* accounts in the database. Users log in to the database using a shared *database* account, retrieve the *software* account information and determine if they have access to the software. Chip designers, database level and project level permissions are also stored in the database.

We defined different database permissions allowing users to add or remove users or projects. A special permission is reserved for the database/software admins, so they have full access to all projects. This means that they can do anything on any projects, even if they are not in the designer list of those projects. We predefined two database roles, the admin and the standard user. However, we can extend the database roles without modifying the software. Similarly, we defined different project permissions allowing chip designers to add system blocks, remove the system blocks they are in charge of, read all system blocks, add or remove chip designers. An implicit permission is that designers can always edit the system blocks they are responsible for.

To make use of the database and project permissions, we designed an **Authenticator** class as in Code 7.13. To make block level authentication more manageable, we also defined block permissions here. They are determined by project permissions and whether the user is the responsible person of the system block. The **Authenticator** class allows users to set database permissions, project permissions and block permissions. The permissions are stored in member variables **db_permissions_**, **project_permissions_**, **block_permissions_** of data type **int**. Each bit of the permissions variables represents a specific permission, which is defined in the enumeration **DATABASE_PERMISSIONS**, **PROJECT_PERMISSIONS**, and **BLOCK_PERMISSIONS**. We set or retrieve permissions using bit operations. For example, to set the **ADD_USER** permission, we

make the 0th bit of **db_permissions_** be 1. To retrieve the **ADD_USER** permission, we get the value of the 0th bit of **db_permissions_**.

Code 7.13: Definition of the Authenticator

```
1  class Authenticator
2  {
3  public:
4      enum DATABASE_PERMISSIONS
5      {
6          ADD_USER = 1 << 0,
7          // ...
8          FULL_ACCESS_TO_ALL_PROJECTS = 1 << 4
9      };
10     // enum PROJECT_PERMISSIONS
11     // enum BLOCK_PERMISSIONS
12
13     Authenticator(const QString& db_role_id, const QString& ↩
           ↪ project_role_id);
14     Authenticator();
15
16     void set_database_permissions(const QString& db_role_id);
17     void set_project_permissions(const QString& ↩
           ↪ project_role_id);
18     void set_project_permissions(bool setting);
19     void set_block_permissions(bool setting);
20     void freeze(bool frozen=true);
21
22     // database permissions
23     bool can_add_user() const;
24     // ...
25     bool can_fully_access_all_projects() const;
26
27     // project permissions
28     bool can_add_block() const;
29     // ...
30     bool can_fully_access_all_blocks() const;
31
32     // block permissions
33     bool can_add_signal() const;
34     // ...
35     bool can_edit_document() const;
36
```

```
37      bool frozen() const;
38
39      void clear_database_permission();
40      void clear_project_permission();
41      void clear_block_permission();
42      void clear_all_permission();
43
44  private:
45      int db_permissions_ = 0, project_permissions_ = 0, ↩
            ↪ block_permissions_;
46      bool frozen_;
47  };
```

We provide a convenient way for managing user accounts using the **UserManagementDialog**. It allows database admins to add or remove users. To add a user, we need a **CreateUserDialog**. The username and initial password are specified and a database role is selected. Like the editor dialogs, the **CreateUserDialog** has to check whether the username is valid. After creation of a user account, users can then log in to the software using the account with the initial password. Users can change their passwords easily using the **ChangePasswordDialog**. To remove a user, however, will be tricky. The users to be deleted might be the owner of some chips. They might be a designer in some chips, and responsible for some system blocks. In this case, our solution is to replace the users in those chips with the database admin who is deleting these users.

On the project level, designers can be added to a chip with the **EditChipDesignerDialog**, where a user and a project role are selected. Before adding a designer, the dialog shall check whether the selected user already exists in the current project. We can also delete a chip designer. Like removing a user, the software has to replace the designers in this project with a project admin.

As previously described, the software account and database account are not the same. In our solution, all software users share a single database account. The problem is that we don't want users to know the password to the database to prevent them from bypassing the software and directly work on the database. This might cause serious data security problems.

Our solution is to introduce an encryptor and decryptor. We designed an encryption class using the AES algorithm [18]. The encryptor can encrypt a plain password with a key. With this key and encrypted password, the decryptor can generate the original plain password. Only the admins know the password. They generate encrypted password, and give users the key and the encrypted password instead of the original password. We made a tool for admins to make encrypted passwords easily as shown in Figure 7.12.

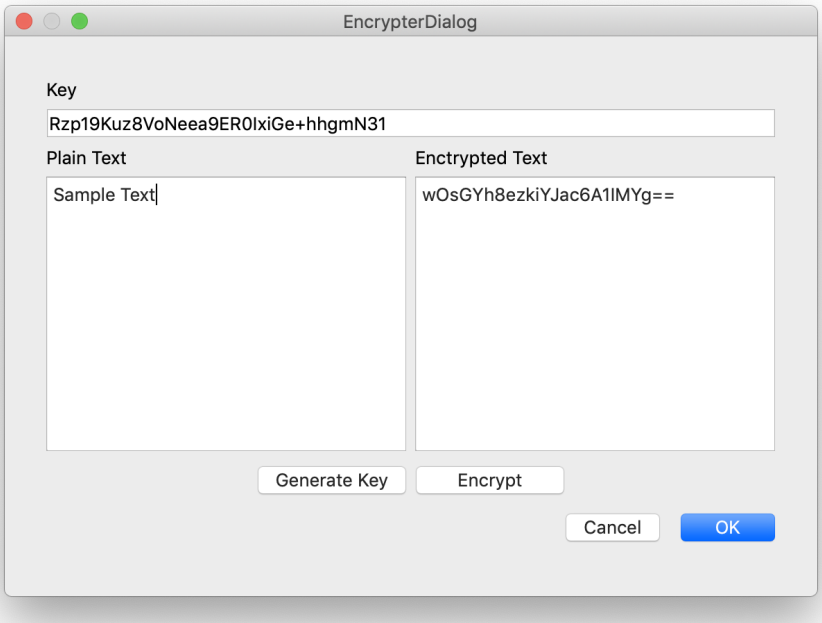


Figure 7.12: Encryptor Dialog

7.4 Signal and Register Naming

In practice, we name signals and registers after a certain pattern. For example, a register might be named after **${BLOCK_ABBREVIATION}_MAIN_REG**. In this example, the block abbreviation is prepended to the central name **MAIN**, and a constant string **REG** is appended to the name. The block abbreviation, what we call the given name, and the suffix **REG** are concatenated with an underscore.

The problem is that, if we have to change a system block's abbreviation, then all registers have to be renamed. This leads to more database operations and unreliability. Our solution is simple: we only store the given name in the database. However, the extended name is generated with the naming template and the given name.

Thus, we designed a **NamingTemplate** class to generate the extended name with the given name, and the other way around. The idea is that we maintain a naming template like **${BLOCK_ABBREVIATION}_${GIVEN_NAME}_REG**. The naming template contains variables like **${Variable}** and constants. The **${GIVEN_NAME}** variable must always be in the template. In this example, to generate the extended name, we simply replace the variables with the abbreviation name of the current system block, and the given name of the current register, which is retrieved from the database. Similarly, we can also extract the given name given an extended name.

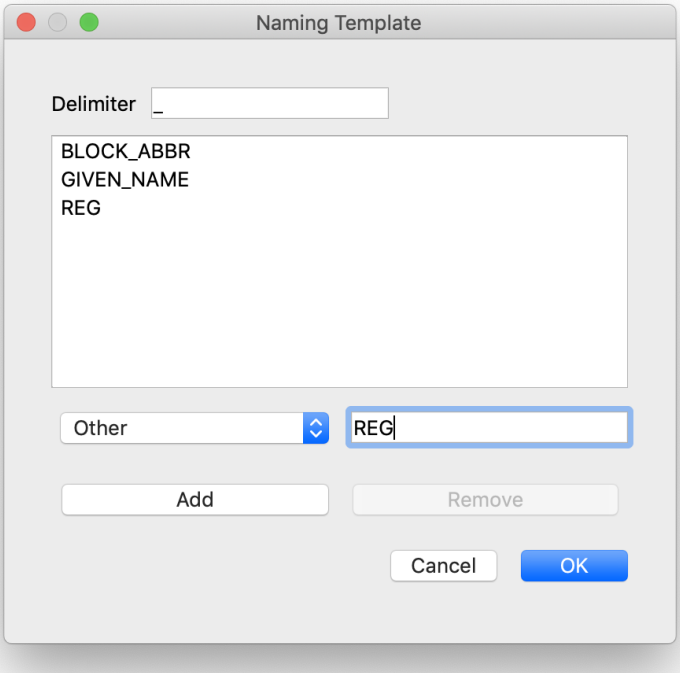


Figure 7.13: Naming Template Dialog

To change the naming of signals or registers, we don't have to do anything on the database.

We simply make a new naming template and refresh the UI. We designed a very friendly dialog in Figure 7.13 to change the naming template. Users just need to add keywords or constants, and specify a delimiter.

7.5 SPI Interface Generation

With all the chip definitions and document items, our final goals are generating the SPI configuration interface and a LaTeX documentation. We might want to generate the interface in different languages later, but so far we only implemented VHDL. Despite of this, during design and implementation, we already left flexibility for extensions.

The VHDL SPI interface consists of two files, the interface package in which basic information about the chip is defined, and the interface in which the VHDL interface and behavior are defined. Code 7.14 and Code 7.15 are an example of the package and interface templates. Users can define a package and an interface template with predefined markers. The software will replace the markers with what it generates from the database.

Code 7.14: VHDL Package Template

```
1  -- sample VHDL package template
2  LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    USE ieee.numeric_std.all;
5
6  PACKAGE interface_package IS
7
8    constant address_width      : integer := 12;
9    constant byte_size          : integer := 8;
10   constant reg_width           : integer := 8;
11   -- FIFO
12   constant byte_counter_width : integer := 10;
13   constant fifo_size          : integer := 2**↩
       ↪ byte_counter_width;
14   constant packet_size        : integer := 2**byte_size;
15
16    -- REGISTER ADDRESSES
17   @PACKAGE_ADDRESSES    -- to be completed by EDA tool
18
```

```
19    -- REGISTER_INITIAL VALUES
20    @PACKAGE_INITS    -- to be completed by EDA tool
21
22  END PACKAGE interface_package;
```

In the package template there are two makers, one for register address definitions, and the other for initial values definitions for writable registers. Since we allocated each system block a start address, it's easy to infer the address for each register. The initial values of each writable registers are not straightforward. We have to compute initial values for each register bit by looking for the corresponding signal bit it is mapped to, and the initial signal value. Then, the initial values of the registers are obtained by concatenating the initial values of each register bit.

Code 7.15: VHDL Interface Template

```
1   -- sample VHDL package template
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use IEEE.NUMERIC_STD.ALL;
5
6   LIBRARY work;
7   use work.interface_package.ALL;
8
9   entity interface is
10    generic
11    (
12      data_width : integer := 12
13    );
14    port
15    (
16      resetn                    : in  std_logic;
17      clk_main                  : in  std_logic;
18      enable                    : in  std_logic;
19
20      -- REGISTER SIGNAL PORT DEFINITIONS
21      @PORT_DEFINITIONS
22
23    );
24  end interface;
25
26  architecture Behavioral of interface is
```

```
27
28    -- INTERNAL SIGNALS FOR INTERFACE FUNCTION
29    signal addr_buffer            : std_logic_vector(↩
         ↪ address_width-1 downto 0);
30    signal reg_read_buffer        : std_logic_vector(byte_size↩
         ↪ -1 downto 0);
31    signal byte_ext2asc           : std_logic_vector(byte_size↩
         ↪ -1 downto 0);
32
33    -- INTERNAL SIGNALS NOT CREATED BY THE AUTOMATIC EXPORT
34    signal SM_INVERT_FIFO_CLK     : std_logic;
35    signal SM_FIFO_SPI_RESETB     : std_logic;
36    signal CHIP_ID_VALUE      : std_logic_vector(16-1 downto 0)↩
         ↪ ;
37    signal CHIP_REGPAGE_CTRL  : std_logic_vector(2-1 downto 0);
38
39    -- REGISTER DEFINITIONS
40    @REGISTER_DEFINITIONS
41
42  begin
43
44    reg_write_proc: PROCESS(clk_main, resetn, enable)
45    BEGIN
46      if resetn = '0' then
47        -- Registers
48        @REGISTER_INIT
49
50        new_spi_reg_write       <= '0';
51      else
52        if rising_edge(clk_main) and enable = '1' then
53          if current_state = S_DATW and byte_ext2asc_rising = ↩
               ↪ '1' then
54            new_spi_reg_write  <= '1';
55            case addr_buffer is
56              @REG_WRITE_ACCESS
57
58              when others                           => null;
59            end case;
60          else
61            new_spi_reg_write <= '0';
62          end if;
63        end if;
64      end if;
```

```
65    END PROCESS reg_write_proc;
66
67    reg_read_proc: PROCESS(clk_main, resetn, enable)
68    BEGIN
69      if resetn = '0' then
70        reg_read_buffer  <= (others => '0');
71        new_spi_reg_read            <= '0';
72      else
73        if rising_edge(clk_main) and enable = '1' then
74          if current_state = S_DATR then
75            new_spi_reg_read        <= '1';
76            case addr_buffer is
77              @REG_READ_ACCESS
78
79              when others                          => null;
80            end case;
81          else
82            new_spi_reg_read        <= '0';
83            reg_read_buffer         <= (others => '0');
84          end if;
85        end if;
86      end if;
87    END PROCESS reg_read_proc;
88
89    @SIGNAL_ASSIGNMENTS
90
91  end Behavioral;
```

The interface is more complex. It consists of two code blocks, the entity definition, and the behavioral definition. In the entity definition, the software shall complete the port definitions. All port signals should be included. Then, in the behavioral definition, the software first completes the register definitions. Then, it should complete the read and write procedure. In the reading procedure, data is read from external and written to a certain register according to the address buffer. In the write procedure, data is read from a certain register and written to external. Finally, the software should assign values to signals or registers. In case of a read-only register, the software shall assign to it values from the signals it is mapped to. In case of a writable register, the software shall assign values to the signals it is mapped to. Paged registers are dealt with in a special way.

We designed a **VHDLGenerator** class. The generation procedure of the interface is described with the Code 7.16. Generation of the package follows the same structure.

Code 7.16: Logic of VHDL Interface Generation

```
// vhdl_generator.h
QString VHDLGenerator::generate_interface() const
{
    QString ports,
            register_definitions, register_inits,
            register_write_accesses, register_read_accesses,
            readonly_register_assignments,
            control_signal_assignments,
            paged_register_assignments;

    QVector<QString> blocks = read_system_blocks();
    for (const auto& block : blocks)
    {
        QVector<QString> interface_block = ↩
           ↪ generate_interface_block(block);
        ports += interface_block[0];
        register_definitions += interface_block[1];
        register_inits += interface_block[2];
        register_write_accesses += interface_block[3];
        register_read_accesses += interface_block[4];
        readonly_register_assignments += interface_block[5];
        control_signal_assignments += interface_block[6];
        paged_register_assignments += interface_block[7];
    }
    QString interface = get_interface_template();
    interface.replace(marker_ports, ports);
    interface.replace(marker_register_definitions, ↩
       ↪ register_definitions);
    interface.replace(marker_register_init, register_inits);
    interface.replace(marker_register_write, ↩
       ↪ register_write_accesses);
    interface.replace(marker_register_read, ↩
       ↪ register_read_accesse);
    interface.replace(marker_signal_assignment, ↩
       ↪ signal_assignments);

    return interface;
}
```

```
34
35  QVector<QString> VHDLGenerator::generate_interface_block(↩
        ↪ block) const
36  {
37      QVector<QString> signals = read_signals(block);
38      QVector<QString> registers = read_registers(block);
39      QString ports,
40              register_definitions, register_inits,
41              register_write_accesses, register_read_accesses,
42              readonly_register_assignments,
43              control_signal_assignments,
44              paged_register_assignments;
45
46      for (const auto& register : registers)
47      {
48          register_definitions += generate_register_definition(↩
                ↪ register);
49          register_read_accesses += generate_reading_register(↩
                ↪ register);
50          if (writable(register))
51          {
52              register_inits += generate_initializing_register(↩
                    ↪ register);
53              register_write_accesses += ↩
                    ↪ generate_writing_register(register);
54          }
55          else
56          {
57              readonly_register_assignments += ↩
                    ↪ generate_assigning_value_to_readonly_register↩
                    ↪ (register);
58          }
59          if (paged_register(register))
60              paged_register_assignments += ↩
                    ↪ generate_assigning_value_to_paged_register(↩
                    ↪ register);
61      }
62
63      for (const auto& signal : signals)
64      {
65          if (port_signal(signal))
66              ports += generate_port_definition(signal);
67          if (control_signal(signal))
```

```
68                 control_signal_assignments += ↩
                      ↪ generate_assigning_value_to_constrol_signal(↩
                      ↪ signal);
69         }
70
71         return {ports,
72                 register_definitions, register_inits,
73                 register_write_accesses, register_read_accesses,
74                 readonly_register_assignments,
75                 control_signal_assignments,
76                 paged_register_assignments};
77   }
```

We made a **SPIGenerationDialog** class for users to configure the export. See Figure 7.14. The design pattern is similar to the previous editor dialogs. Since the generated code must be directly usable, we have to make sure there are no errors. Sanity checks must be done.

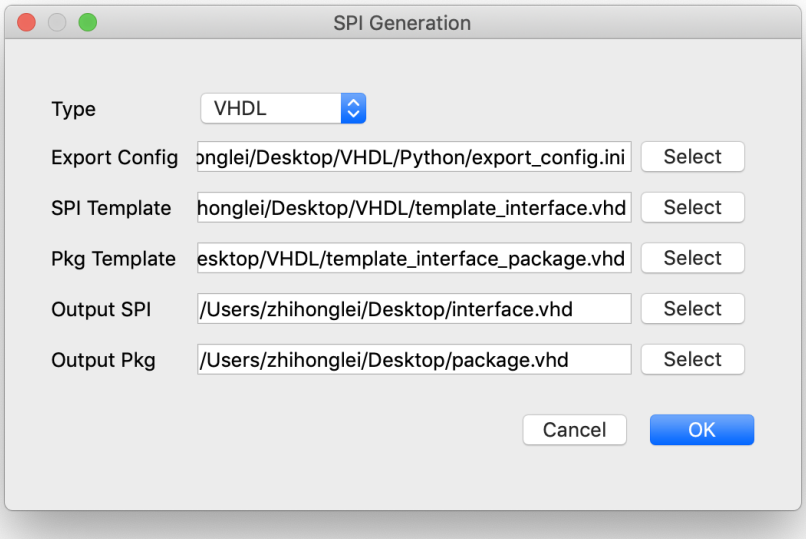


Figure 7.14: SPI Generation Dialog

7.6 Document Generation

So far chip designers have added documents to each system block, register and signal, and also the chip itself as well. The software will compose them and generate a complete documentation.

The documentation has a good CHIP-BLOCK-REGISTER-SIGNAL structure as in Figure 7.15. Thus, we designed a **DocumentGenerator** following this structure. Code 7.17 shows the definition of the **DocumentGenerator** class. For simplicity, function parameters and member variables are omitted.

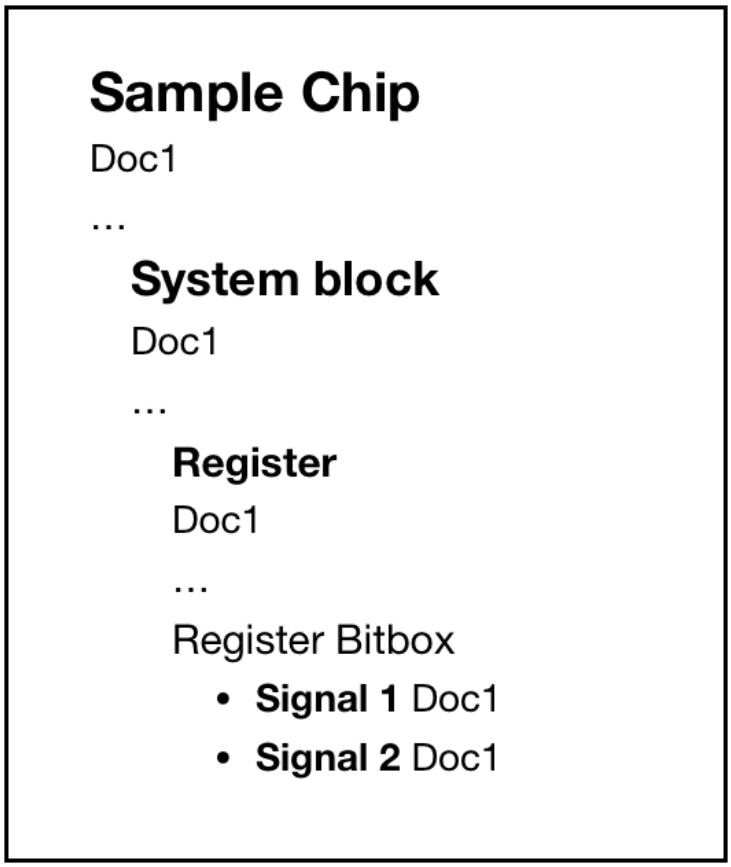


Figure 7.15: Structure of the Documentation

Code 7.17: Definition of the Document Generator

```
// document_generator.h

// functions to generate the LaTeX documentation
// parameters are omitted
QString generate_tex_document();
QString generate_chip_level_tex_document();
QString generate_block_level_tex_document();
QString generate_register_level_tex_document();
QString generate_register_bit_table_tex();
QString generate_register_signal_bullets_tex();

// static functions to generate single documents into LaTeX
static QString generate_text_tex(const QString& text);
static QString generate_image_tex(const QString& caption, ↩
    ↪ const QString& width, const QString& path);
static QString generate_table_tex(const QString& caption, ↩
    ↪ const QVector<QVector<QString> >& cells);
```

In previous sections, we have introduced a document preview for the document editor. It is basically a web browser rendering HTML web pages with LaTeX source code with the help of MathJax. Since the software is already able to display a single document, it is pretty straightforward to implement a function to generate a full documentation in HTML format, and make a preview of the complete documentation based on MathJax and web browsers. Figure 7.16 is an example of the documentation preview using our software. In fact, in some cases HTML would be a better choice than LaTeX, for example, if the Chair of IAS wants to create an internal wiki about the ASIC chips.

We created a **DocumentGenerationDialog** as in Figure 7.17 for users to select the document format, configure the documentation and select which system blocks they want to include in the documentation.

Document Preview

SampleChip

Global

GLOBAL_RESETB1_REG - 0x000

7	6	5	4	3	2	1	0
...	WUR_RESETB	OSCI_CTRL_RESETB	ULP_RESET	SM_RESETB	TX_RESETB	DEM_RESETB	PLL_RESETB
.	0	0	0	0	0	0	0

- WUR_RESETB is written to WAKEUP<3>
- OSCI_CTRL_RESETB is written to OSCI_CTRL<0>
- ULP_RESET is written to ULP_MAIN<1> and inverted to to ULP_MAIN<4>
- SM_RESETB is written to SM_MAIN<1>
- TX_RESETB is written to TX_MAIN<5>
- DEM_RESETB is written to DEM_MAIN<7>
- PLL_RESETB is written to PLL_MAIN<3>

GLOBAL_RESETB2_REG - 0x001

7	6	5	4	3	2	1	0
...			RXADC_RESETB	BBPLL_RESETB	OTP_RESETB	RSSI_CTRL_RESETB	SLEEP_VREF_RESETB
.	.	.	0	0	0	0	0

- RXADC_RESETB is written to RXADC_SET_REG<7>
- BBPLL_RESETB is written to BBPLL_CTRL_REG<7>
- OTP_RESETB is written to OTP_CONFIG_REG<0>
- RSSI_CTRL_RESETB is written to RSSI_CTRL_REG<3>
- SLEEP_VREF_RESETB is written to SLEEP_VREF_REG<2>

Figure 7.16: Document Preview in HTML Format

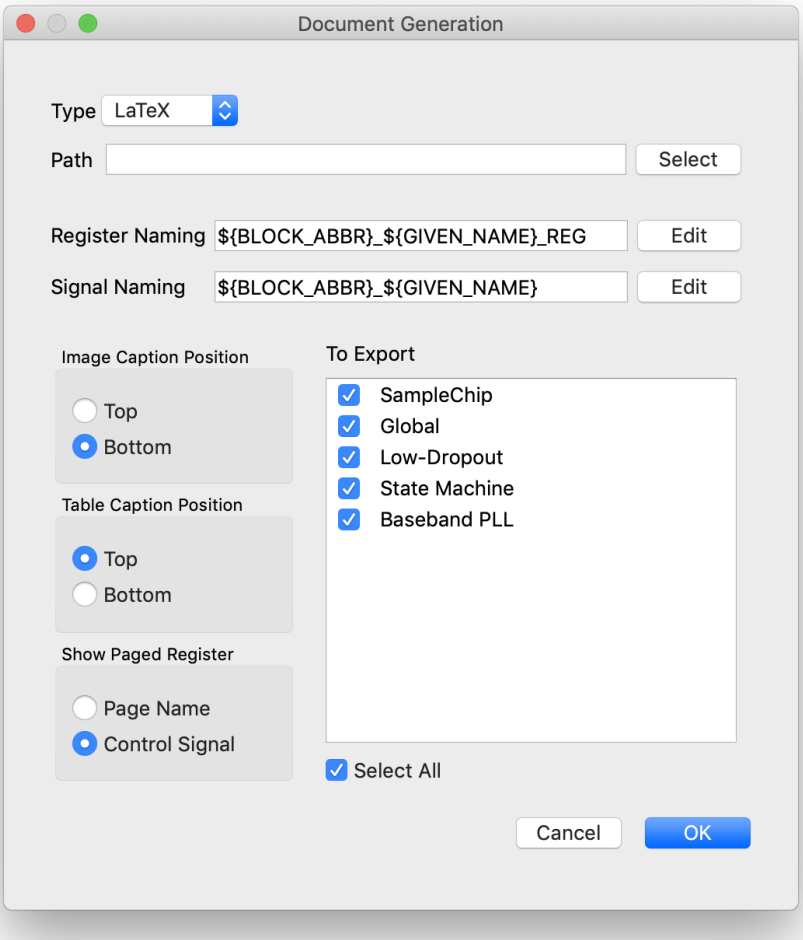


Figure 7.17: Documentation Generation Dialog

8 Integration, Testing and Deployment

So far we have completed the design and implementation process. During implementation, we adopted an incremental strategy. We tested each module right after it is implemented. Now, we want to integrate the modules and test the software as a whole.

Therefore, we made a beta version and delivered it to the users, which allows them to participate in the testing phase by using the software solution on real-world projects. During this process, some errors were reported and we were able to solve them. Users also gave suggestions on the software and we were able to improve it so as to better satisfy users' requirements.

To test the software, users have to install a MySQL C++ connector and the whole Qt framework on their machines. Then, they have to configure the linked libraries and compile the software locally. This is not so straightforward. We still have to make a deployable software so that users don't have to install the dependencies by themselves.

The theory is that the external dependencies including the Qt Modules and MySQL connector are not incorporated in our source code. Instead, they are used in form of shared libraries. During compilation, the executable knows where to find the shared libraries it depends on. During execution, the executable finds the libraries and loads them into the computer memory. This is called dynamic linking. If the executable cannot find the libraries it needs, it will crash.

Thus, we need to manually collect all Qt frameworks and the MySQL connector library the software depends on. We will then package the libraries with the executable under

the same directory, and make the executable find the libraries under this directory during runtime. We can either specify an **rpath** for the linker during compilation, or relink the executable after it is compiled.

It is not easy to manually deploy the software. Fortunately, there are tools helping us with deployment. The Mac OS version of Qt itself provides a tool called **macdeployqt**. It is very simple to use. We just need to compile the software as usual. Then, we use **macdeployqt** to deploy the software. We only need to specify the path to the executable. The tool will do everything for us: make a software directory, find the dependency libraries, copy and paste those libraries under the directory, and relink the executable.

To deploy the software on a Linux system, although Qt does not provide a tool like **macdeployqt**, the community developed a tool named **linuxdeployqt** [19] which basically follows **macdeployqt**. With the help of **linuxdeployqt**, we successfully deployed the software to the Chair of IAS.

Bibliography

[1] Ian Sommerville. *Software engineering 9th Edition*. Addison-wesley, 2011.

[2] Steve McConnell. *Rapid development: taming wild software schedules*. Pearson Education, 1996.

[3] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[4] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[5] *What is Copyleft?* https://www.gnu.org/licenses/copyleft.en.html.

[6] *PyQt*. https://riverbankcomputing.com/software/pyqt/intro.

[7] *MySQL*. https://www.mysql.com.

[8] *MariaDB*. https://mariadb.org.

[9] Mark Richards. *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[10] Bruce Eckel. *Thinking in JAVA*. Prentice Hall Professional, 2003.

[11] *Qt*. https://www.qt.io.

[12] *Qt Modules*. https://doc.qt.io/qt-5/qtmodules.html.

[13] *Qt Signals and Slots*. https://doc.qt.io/qt-5/signalsandslots.html.

[14] *MySQL Connector C++*. https://dev.mysql.com/doc/dev/connector-cpp/8.0/.

[15] *MathJax*. https://www.mathjax.org.

[16] *Qt Web Engine*. https://doc.qt.io/qt-5/qtwebengine-index.html.

[17] *The Chromium Projects*. https://www.chromium.org.

[18] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[19] *linuxdeployqt*. https://github.com/probonopd/linuxdeployqt.

A Presentation Slides

A.1 First Master Thesis Presentation

Automated Implementation of the Digital Configuration Interface for Application Specific Integrated Circuits

Zhihong Lei

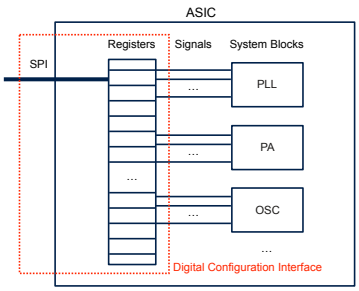
First Master Thesis Presentation

Supervisor: Johannes Bastl, M.Sc

Univ.-Prof. Dr.-Ing. Stefan Heinen
Integrated Analog Circuits and RF Systems Laboratory

Background

Configuration of ASIC System Blocks

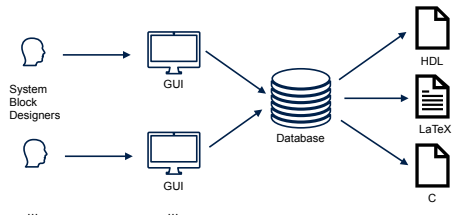


Problems

- Great amount of manual work
  - HDL source code
  - LaTeX documentation
  - Other programming languages (C, MATLAB etc)
- Communication and synchronization
- Complex ASIC: hundreds of registers

Proposed Solution

- A software that automatically generates HDL source code, LaTeX documentation and code in other formats (C, etc)



Advantages

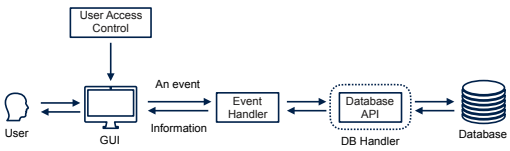
- Human labor will be greatly reduced
- Synchronization between different exports will be intrinsically ensured
- Errors can be eliminated (given that the register definitions themselves are correct)

Crucial Requirements

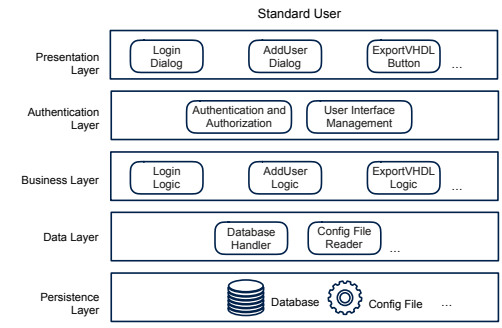
- Reliable storage and management of registers definitions and initial values
- Information security (user access control)
- Reliable export functions
- A friendly graphical user interface (GUI)

Software Design

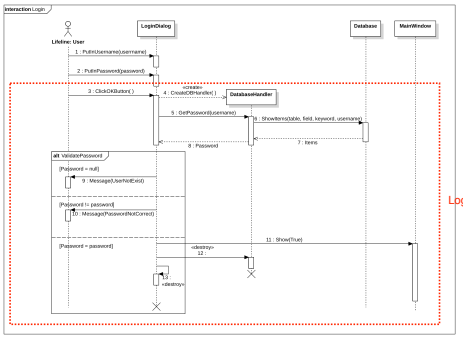
Event-Driven Application Design



Architecture and Module Design



Workflow of the Layered Architecture



Advantages of the Architecture

- Each layer is clearly defined
- Layers are ideally only dependent of neighboring layers
- Modules are well encapsulated and the boundaries are clear

Selection of Infrastructure

Selection of the GUI Framework

Toolkit Name	Cross-platform	Programming Language	Specialized IDE	Licence
Qt	✓	Multiple	✓	LGPL
MFC	✗	C++	✓	Proprietary
GTK	✓	Multiple	✗	LGPL
Swing	✓	Java	✓	-
Tk	✓	Multiple	✗	BSD
wxWidgets	✓	Multiple	✗	WxWindows license

- Qt is a preferred choice

Why Qt?

- Qt Designer - specialized IDE for Qt, with which we can design GUI in a WYSIWYG manner
- Cross-platform
- Free software with a very loose license
- Easy to learn and to use
  - Well organized documentation and a strong developer community
  - Concise, elegant and user friendly APIs

Requirements for Data Storage

- Database shall be accessible to all users (on different local machines)
- Database shall be concurrently read or modified by multiple users
- Data shall be reliably stored
- Data access should be efficient
- Data should be well organized

A well-designed relational database can fulfill these requirements

Selection of the Database Management System

DBMS Name	Cross-platform	License	Client/Server	Comment
MySQL	✓	GPL	✓	Most widely used open-source database system
Microsoft Access	✗	Proprietary	✗	Expensive
SQLite	✓	Public domain	✗	Limited functionalities
Microsoft SQL Server	✗	Proprietary	✓	Expensive
Oracle	✓	Proprietary	✓	Expensive
MariaDB	✓	GPL	✓	Compatible to MySQL

- MySQL or MariaDB are the best choice

Work Schedule

Work Process

- So far
  - Architecture design
  - Selection of GUI frameworks
  - Selection of database systems
  - GUI prototyping (ongoing)
  - Database design (ongoing)
- To do
  - Implementation of software modules
  - Testing and trial run
  - Documentation of the software
  - Thesis writing

Time Table

Timeline	Work
31.03	Requirement engineering; Database structure design finalization
30.04	Presentation and authentication layers finalization; Part of modules on the business layer
31.05	Implementation of LaTex, VHDL and other formats export
30.06	Intensive test and verification; implementation of additional functionalities
31.07	Trial run; documentation
31.08	Thesis writing
30.09	Finalization

Thank you

A.2 Second Master Thesis Presentation

Automated Implementation of the Digital Configuration Interface for Application Specific Integrated Circuits

Zhihong Lei

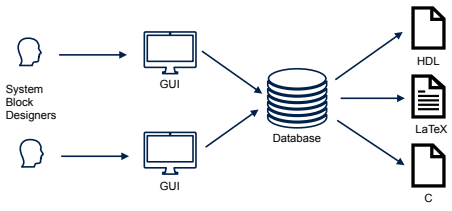
Mid-Term Master Thesis Presentation

Supervisor: Johannes Bastl, M.Sc

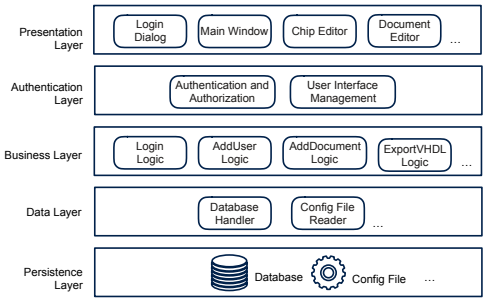
Univ.-Prof. Dr.-Ing. Stefan Heinen
Integrated Analog Circuits and RF Systems Laboratory

Proposal

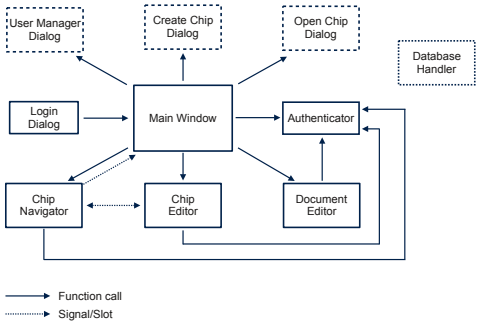
- A database centered software that aids in implementation of digital configuration interface for ASICs



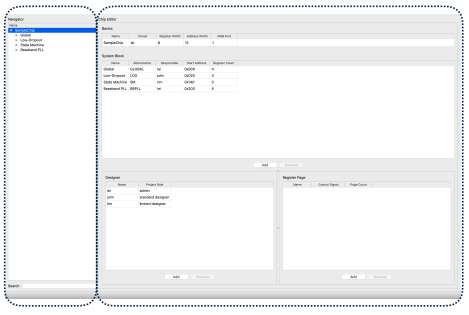
Designed Architecture



Object-Oriented Programming in Practice



Main Window



Chip Navigator Work Area

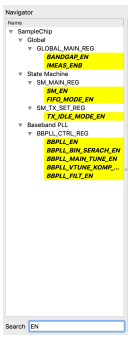
Main Window

Chip Navigator Work Area

Chip Navigator

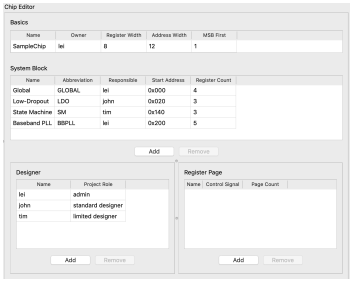
4-level tree widget
chip, block, register, signal

Search area



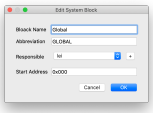
Chip Editor

- Chip-level view



Basic Information about the chip

List of system blocks

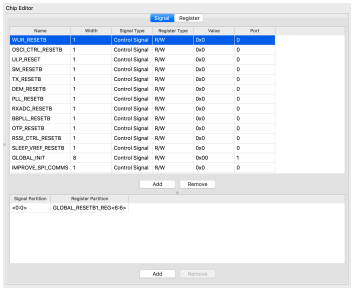


Designers

List of register pages

Chip Editor

- Block-level View - Signals

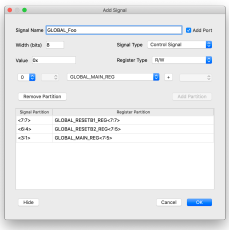


All signals in current block

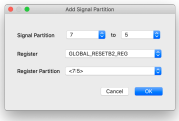
Signal-Register mappings

Chip Editor

- Add/Edit signals



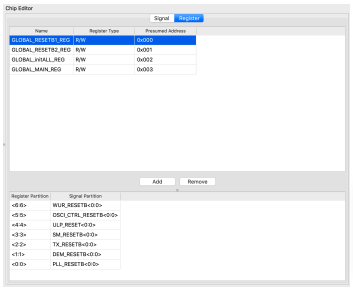
Add/Edit Multi-bit Signal dialog



Add Signal-Register Mapping dialog

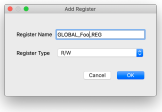
Chip Editor

- Block-level View - Registers



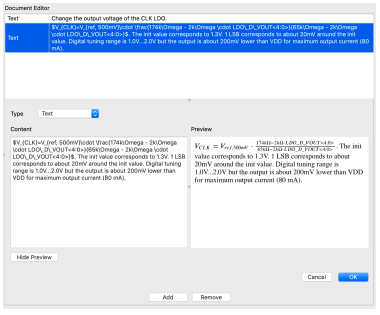
All registers in current block

Register-Signal mappings



Add Register dialog

Document Editor

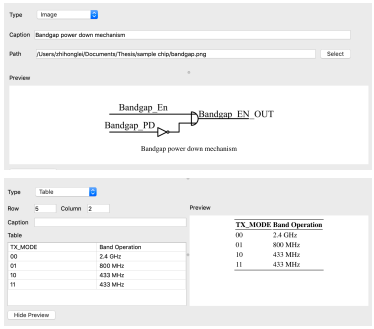


List of documents of currently selected block/register/signal

Document editing area

Preview: HTML web viewer supporting LaTeX

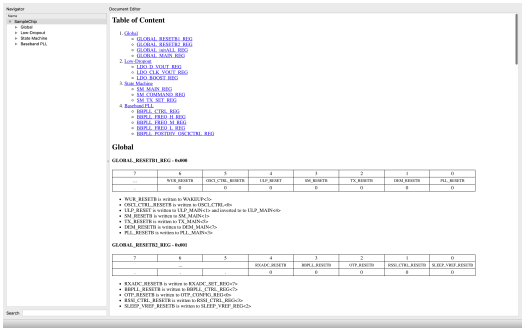
Document Editor



Image

Table

Document Overview



Live Demo …

Time Table

Timeline	Work
15.07	Implementation of export function of LaTeX document and VHDL source code
31.07	Implementation of database exception handling; refining the software
31.08	Deployment; trial run; thesis writing
30.09	Finalization

Thank you

A.3 Final Master Thesis Presentation

Automated Implementation of the Digital Configuration Interface for Application Specific Integrated Circuits

Zhihong Lei

Final Master Thesis Presentation

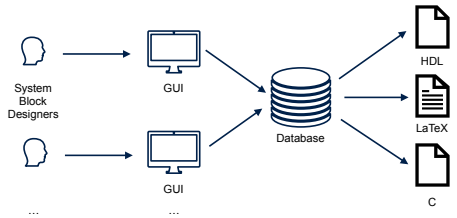
Supervisor: Johannes Bastl, M.Sc

Univ.-Prof. Dr.-Ing. Stefan Heinen
Integrated Analog Circuits and RF Systems Laboratory

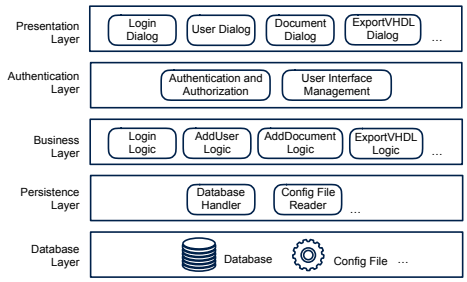
Recap

Proposal

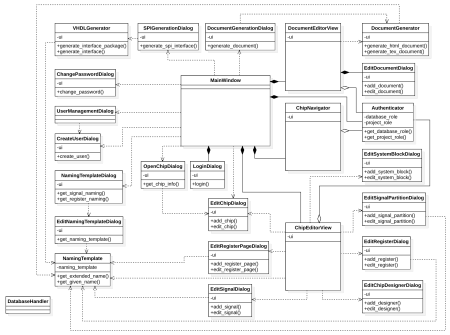
- A database centered software that aids in implementation of digital configuration interface for ASIC chips



Layered Architecture Design

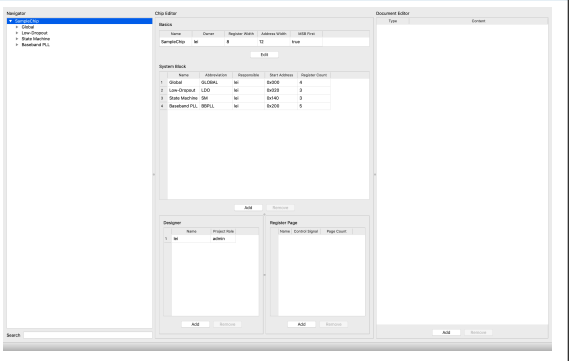


Object-Oriented Software Design



Module Design and Implementation

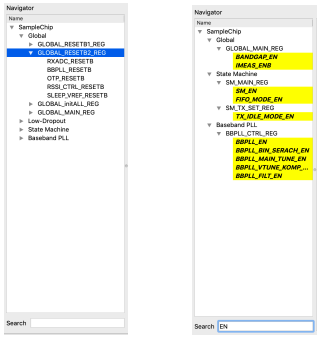
Main Window



Chip Navigator

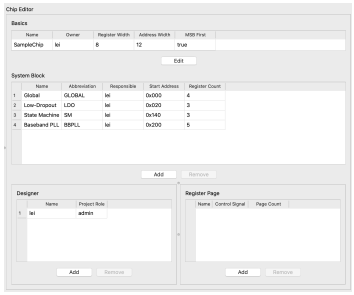
4-level tree widget
chip, block, register, signal

Search area



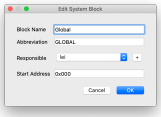
Chip Editor

- Chip-level view



Basic Information about the chip

List of system blocks



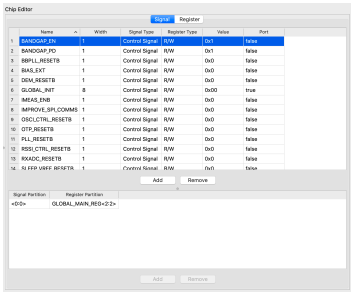
EditSystemBlockDialog

Designers

List of register pages

Chip Editor

- Block-level view - Signal

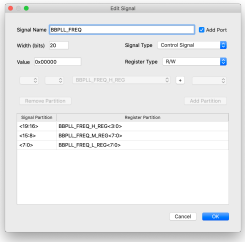


All signals in the current block

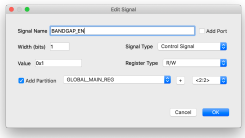
Signal-register mappings

Chip Editor

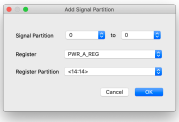
- Block-level view - Signal



EditSignalDialog - Multi-bit



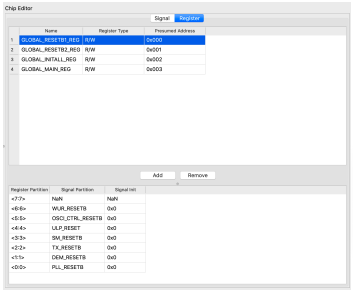
EditSignalDialog - Single-bit



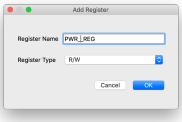
EditSignalPartitionDialog

Chip Editor

- Block-level view - Register



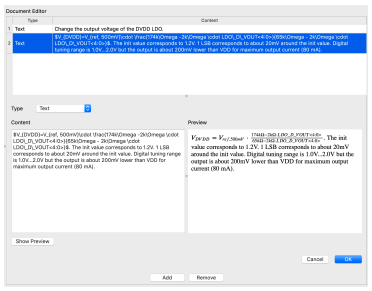
All registers in the current block



EditRegisterDialog

Register-Signal mappings

Document Editor

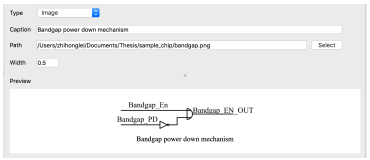


List of documents of currently selected block/register/signal

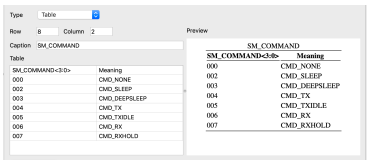
Document editing area

Preview: HTML web viewer supporting LaTeX

Document Editor



Image



Table

Document Editor

- Auto-completion

Text content

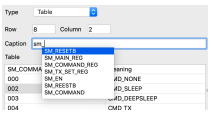


Table & image caption

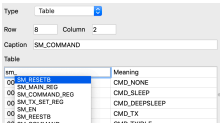
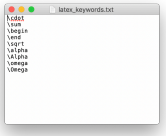


Table cell

Document Editor

- Auto-completion - candidate words
  - Builtin
    - Chip name
    - Block names and abbreviations
    - Register names
    - Signal names
  - User defined
    - ${root}/completion/*.txt



Example word list

Accounts and Passwords

- Database account: shared to all software users

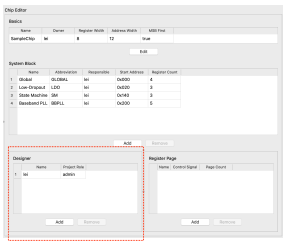
```
# ${root}/global_settings.ini
[database]
database=ias
encrypted_password="TvCYyAGe6hcIyj4ftIg31w=="
host=localhost
key=vFCI399bBoVpu7o89rGbQ/TYyJjOG8ft
port=3306
username=root
```

- Software account

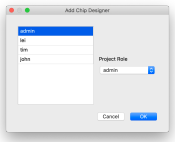
LoginDialog

Accounts and Passwords

- Chip designer account



List of Chip Designers



EditChipDesignerDialog

Permissions and Authentication

- Software/database permissions: software users
  - Add or remove a user account
  - Add or remove a chip project
- Project permissions: chip designers
  - Add system blocks
  - Remove system blocks the user is responsible for
  - Read all system blocks
  - Add or remove chip designers
  - Edit system blocks the user is responsible for (implicitly true for all chip designers)
- Authenticator class

Signal and Register Naming

- Signals and registers may be named after a certain pattern

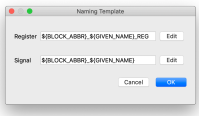
Block Abbr. Given Name Constant

- Question: what if block abbreviations change?
- Solution
  - Only store the given names in the database
  - Define naming templates for signals and registers

    Example: **${BLOCK_ABBR}_${GIVEN_NAME}_REG**
  - Recover the extended name

Signal and Register Naming

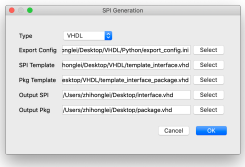
- NamingTemplate class
  - naming_template: string
  - get_extended_name(given_name: string) -> string
  - get_given_name(extended_name: string) -> string
- Naming templates are easily adjustable



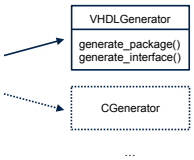
NamingTemplateDialog

EditNamingTemplateDialog

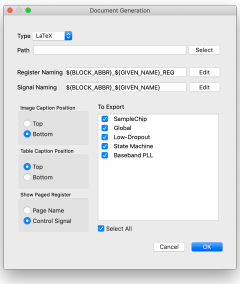
SPI Interface Generation and Export



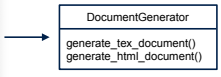
SPIGenerationDialog



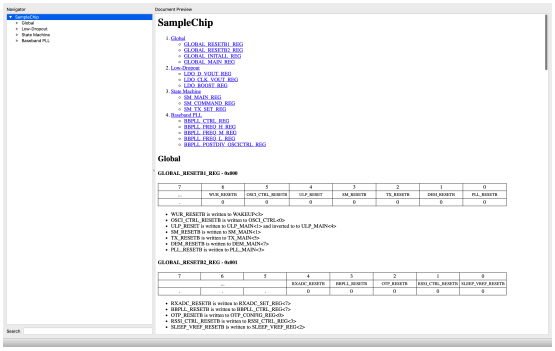
Document Generation and Export



DocumentGenerationDialog



Documentation Preview



Integration, Testing and Deployment

Integration and Testing

- Module testing: right after implementation
- Integrated testing
  - Beta version delivered to users
  - The software was used on real-world projects
  - Errors and bugs were found and removed
  - Some functionalities were improved

Deployment

- The software depends on external softwares in form of shared libraries
  - Qt modules
  - MySQL C++ Connector
- Problem: users have to manually install all dependencies and link them to the executable
- Deployment
  - Package the executable and all dependencies
  - Relink the executable to these dependencies
  - Deliver the software package to users
- We deployed the software using linuxdeployqt tool

Thank you