
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

# BACHELOR'S THESIS

## IP design tasks in a high-tech development

Author Reino Jokitulppo

Supervisor Jukka Lahti

December 2023

**Jokitulppo R. (2023) IP design tasks in a high-tech development.** University of Oulu, Degree programme in Electronics and Communications Engineering, Bachelor's Thesis, 21 p.

# ABSTRACT

**This thesis introduces how a system-on-chip intellectual property is designed. In the beginning a short introduction about SoC IPs is given. In the main chapters the thesis gives a short introduction to every major IP design task. In the later chapter I discuss my own experiences in designing an IP block during my internship at Nokia.**

**Key words: SoC, logic design, RTL, formal verification, IP-Xact, AXI4.**

**Jokitulppo R. (2023) IP suunnittelun tehtävät korkean teknologian suunnittelutyössä.** Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 21 s.

# TIIVISTELMÄ

**Tämä opinnäytetyö esittelee, kuinka tavallinen järjestelmäpiiri IP suunnitellaan. Alussa lyhyt esittely järjestelmäpiiri IP:stä. Pääkappaleissa käydään läpi jokainen tärkeä IP suunnitteluun liittyvä tehtävä. Myöhemmässä kappaleessa käyn läpi omia kokemuksiani IP-lohkon suunnittelusta harjoittelujakson aikana Nokialla.**

**Avainsanat: järjestelmäpiiri, logiikkasuunnittelu, RTL, formaali varmennus, IP-Xact, AXI4.**

# TABLE OF CONTENTS

# FOREWORD

This thesis was done as a part of my internship at Nokia. The purpose was to examine the tasks included in designing an IP block in a high-tech company. Thanks to Saurav Bhaumick and Kari Rysberg for guiding me through this writing process during my internship. Thanks to Jukka Lahti for supervising this thesis.

Oulu 5.12.2023

Reino Jokitulppo

# EXPLANATIONS OF ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| IP | Intellectual Property |
| SoC | System-on-Chip |
| MUX | Multiplexer |
| ALU | Arithmetic Logic Unit |
| CPU | Central Processing Unit |
| HDL | Hardware Description Language |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| XML | Extensible Markup Language |
| EDA | Electronic Design Automation |
| VIP | Verification Intellectual Property |
| AIP | Assertion Intellectual Property |

# 1 INTRODUCTION

Intellectual Properties (IPs) are used to ensure efficient SoC development. Using these pre-designed and pre-verified components saves time, ensures reliability, and accelerates time-to-market.

Designing an IP block usually follows a predefined sequence, which includes various tasks and tools. The purpose of this thesis made for Nokia is to examine tasks included in designing an IP block. The thesis is split into two parts: In the first part, each design task is gone through briefly. During this first part, a deeper look at one communication interface is taken, because examining different communication interfaces was a part of my internship at Nokia. In the second part, my experiences and thoughts in designing an IP block as a part of a project at Nokia is discussed.

# 2 IP DESIGN TASKS

In the field of SoC IP design, a basic design flow is usually followed. Some steps happen in parallel and some in sequentially. Figure 1 below illustrates this. Following this kind of predefined flow saves time, improves efficiency, reduces risks and enhances collaboration. This minimizes the cost of development, which is a key factor in all SoC IP development. [1].

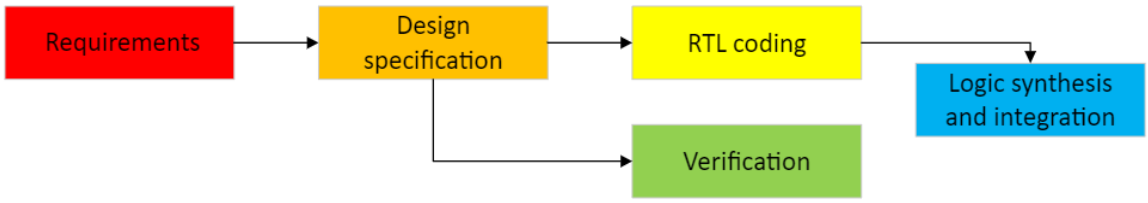


Figure 1. SoC IP design tasks

## 2.1 Design Specification

When designing the IP, there are many factors which must be considered during the design specification stage. The designer needs to know which kind of bigger entity will the newly created IP be integrated? IP blocks communicate between themselves via different communication protocols called bus interfaces. Choosing a widely used industry-standard bus interface, for example Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) protocol makes the design easy to integrate later. As this bus interface is widely used in the industry, it is examined more as an example in the following chapter. Also, the internal functionality of the IP block is specified during the architecture specification stage.

### *2.1.1 Advanced eXtensible Interface bus protocol*

AMBA AXI is a bus protocol developed by ARM. There are currently 3 versions of AXI protocol, AXI3, AXI4 and AXI5. We focus on AXI version 4 here. AXI4 is widely used among SoC field due to its versatility for many applications. It is used mainly for its high bandwidth and low latency. AXI4 is a point-to-point communication protocol with a manager-subordinate interface. AXI4 protocol consists of five independent channels, as illustrated in figure 2.

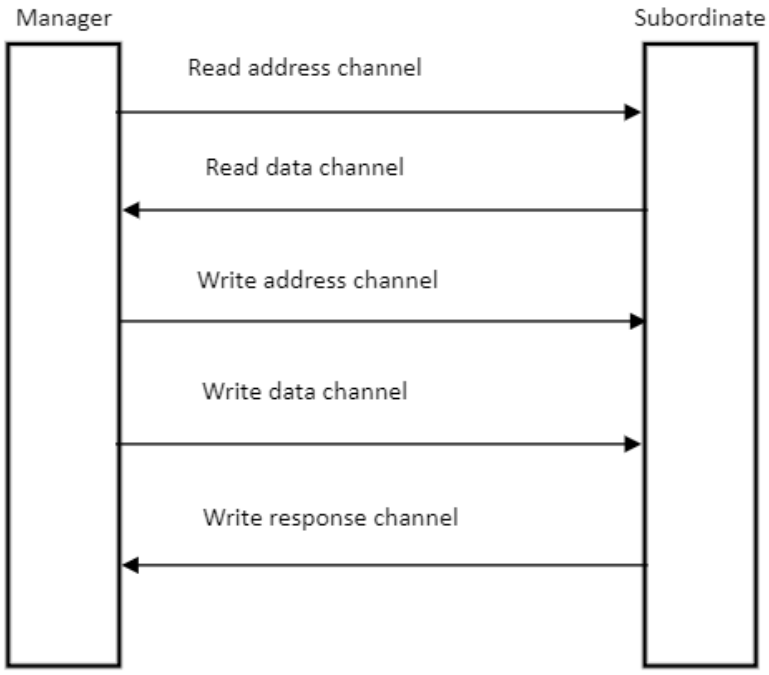


Figure 2. AXI4 channel architecture

The data is transferred using either write data channel or read data channel. AXI4 uses a basic handshake mechanism which is based on two signals, xVALID and xREADY. Each of these 5 channels have their own VALID and READY signal. Manager uses VALID signal to show that valid address, data or control information is ready. The subordinate uses READY signal to indicate it's ready to receive the data. AXI4 contains many signals, and a list of those can be found from appendix 2. The AXI4 handshake mechanism is illustrated in figure 3.[9]

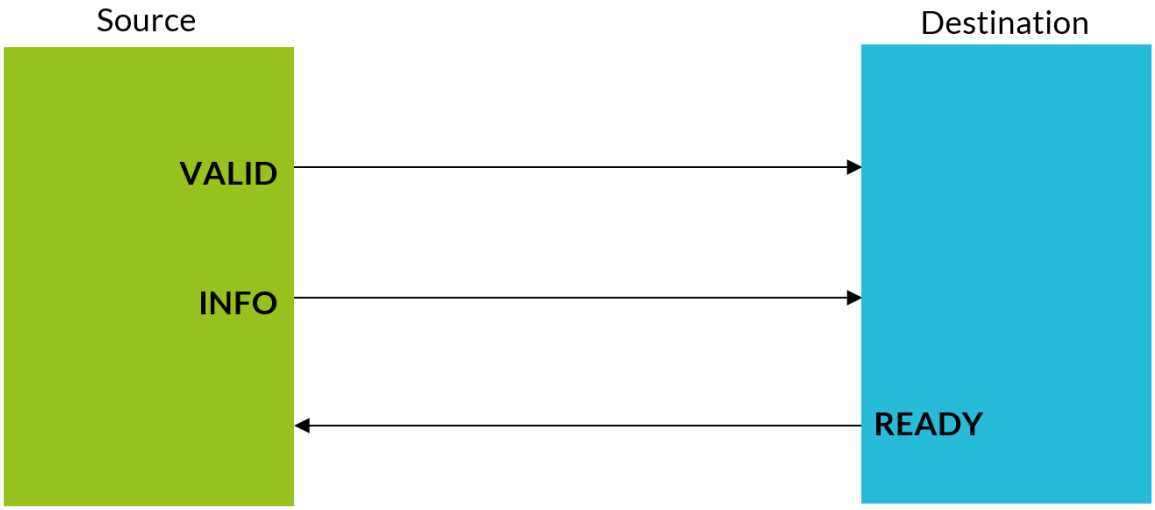


Figure 3. AXI4 handshake

### *2.1.2 Internal functionality*

During the design specification, the designer needs to implement the functionality on a block-level. The block-level design usually constist of multiplexers (muxes), registers and other combinational or sequential blocks. Combinational logic is a type of logic which output is a function of its inputs. A simple example of a combinational block is the arithmetic-logic unit (ALU) on a central processing unit (CPU) core. Sequential logic is a type of logic which output depends on the current value of its inputs and on the past inputs. Basically, this means that sequential logic has memory, while combinational logic doesn't. A simple example of sequential logic would be, for example, a counter circuit. Figure 4 illustrates a block level diagram of this earlier described ALU.

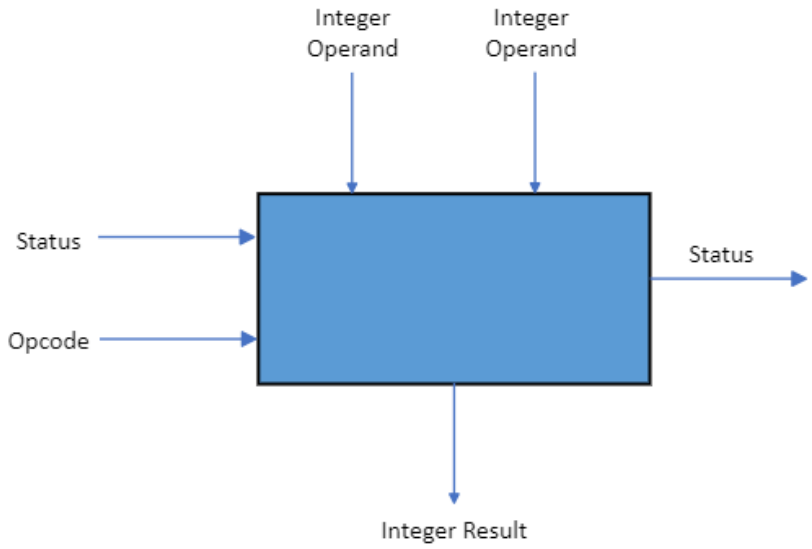


Figure 4. Arithmetic logic unit

Internal functionality is usually specified on a block level diagram. Block level diagram gives good visual understanding about the IP's inputs and outputs. It is easy for the designer to decide which submodules they encapsulate in a single hardware description language (HDL) file.

## 2.2 RTL Coding

Digital logic circuits are usually described on register transfer level (RTL). Register transfer level means that the signals (data) flows between hardware registers, and some logic operations are done to these signals between the registers. RTL digital circuits are described using HDLs like SystemVerilog and VHSIC Hardware Description Language (VHDL). In this RTL coding chapter, the thesis focuses on using SystemVerilog to model synthesizable digital logic.

SystemVerilog is an extension of Verilog. SystemVerilog has three procedural blocks that are meant to model hardware: always_comb for modeling previously described combinational

logic, always_ff to model flip-flops and always_latch for latches. When designing an IP, the written HDL code must be synthesizable. This means that a synthesis tool can translate the RTL code into logic gates. Figure 5 is a code snippet for synthesizable 4 to 1 multiplexer in SystemVerilog. Notice the always_comb -procedure. Figure 6 shows the SystemVerilog code for D flip-flop. Note the always_ff -procedure [3].

```
1  always_comb begin : mux4to1
2    unique case (select)
3      2'b00: data_o = data_i[0];
4      2'b01: data_o = data_i[1];
5      2'b10: data_o = data_i[2];
6      2'b11: data_o = data_i[3];
7    endcase
8  end : mux4to1
9
```

Figure 5. 4 to 1 multiplexer.

```
1  always_ff @(posedge clk) begin : flip_flop
2    if (~resetn)
3      data_out <= 0;
4    else
5      data_out <= data_in;
6  end : flip_flop
```

Figure 6. D flip-flop.

When writing RTL code, the designer can use various methods to make sure the RTL code is synthesizable. The most common way of preventing a non-synthesizable code is to follow the synthesizable coding rules and to use so-called linting tools. Linting tools are code-checking tools that report bugs, errors and even style issues.

### 2.3 Verification

In a high-tech development the SoC designs are so complicated and huge that one cannot make sure the design works as intended by just looking at the pure HDL code. If the desired functionality is not met and the design passes into fabrication stage, the financial costs could be millions of euros. This is why the design must be verified with many different methods to ensure a bug-free and perfectly working SoC. One could argue that the purpose of verification is to find bugs rather than prove its correct functionality. In this chapter, the thesis only focuses on a very small part of the verification, because the verification is a huge topic, and this chapter will only cover a small fraction of it.

IP verification relies mainly on white-box verification. In white-box verification the IP's internal structure is known, so the engineer writing the test program can, for example, refer to the IP block's internal variables. This is in contrast to black-box verification, where the IP to be tested is not known to the tester. In black-box verification, the tester focuses on examining the input and output signals [5]. White-box verification for IPs uses methods like formal verification and random simulation. [4].

Formal verification is a part of functional verification. Formal verification tries to prove the model's mathematical correctness. One practical example of formal verification is the use of SystemVerilog assertions. SystemVerilog assertions (SVA) is a language construct that makes writing assertion statements easier and faster [7]. These assertions are statements that

check if the design behaves as intended. If the design does not behave as intended, the assertion will fail. There are different types of assertions: Assert, assume, cover and restrict statements, for example [6]. Figure 7 is a code snippet for a simple SVA statement.

```
1  module assert_example;
2    logic a;
3    logic b;
4
5    always_comb begin
6      //functionality
7      assert (a < b) begin
8        $display("a is smaller than b);
9      end else begin
10       $display("a is bigger than b);
11     end
12   end
13 endmodule
```

Figure 7. SVA statement example.

This example shows how easy it is to convert human language sentence to an SVA statement. Code above in human-understandable language would be: ”A must be smaller than b”. This statement tests the possible functionality of an imaginable IP block. The engineer doing the verification doesn’t usually write all the assertions himself. He makes use of Assertion IPs (AIPs). They are ready made file sets which contain assertions. These AIPs make the verification faster, as one doesn’t need to write hundreds of assertions if the IP to be verified has some already known standard features, such as communication interfaces.

Verification is done at different levels of scope. In this thesis we have been focusing on IP level verification. Verification is also done at the sub-system level. Sub-systems contain already verified IPs. At this level, the verification is done by using Verification IPs (VIPs). VIP is a reusable testbench that follows some testbench coding standards. At one hierarchy above sub-system is the entire SoC level. At that level, the verification environment will have all kinds of testbench components, such as UVM verification IPs or for example SystemC functional models. [4]

### 2.4 Logic synthesis and integration

After the IP has gone through verification, the synthesis takes place. In synthesis, the RTL model gets converted to a netlist containing standard library cells. The last IP design task is integrating the newly created IP into a part of a bigger entity. Nowadays integration is made easier by using industry standard formats to store and integrate these IPs.

The logic synthesis has two parts. In the first part, the RTL model gets converted into a generic netlist, and in the second part the generic netlist is converted to a netlist that contains standard cells from the so-called technology library. Standard cell library contains, for example, logic cells like AND, OR, NOR, XOR ports or D flip-flops. The standard cells can also include a bit more complex cells, like multiplexers or adders. The synthesis is done by using a synthesis tool, which uses high-tech optimization algorithms to convert the RTL model into a netlist. The tool is optimized to remove redundant logic while keeping the behaviour the same. In this stage it is important that during the RTL coding phase, the HDL

code is written by using the right HDL constructs. As discussed earlier, this is usually achieved by using lint tools. [1] The synthesis process is illustrated in figure 8.

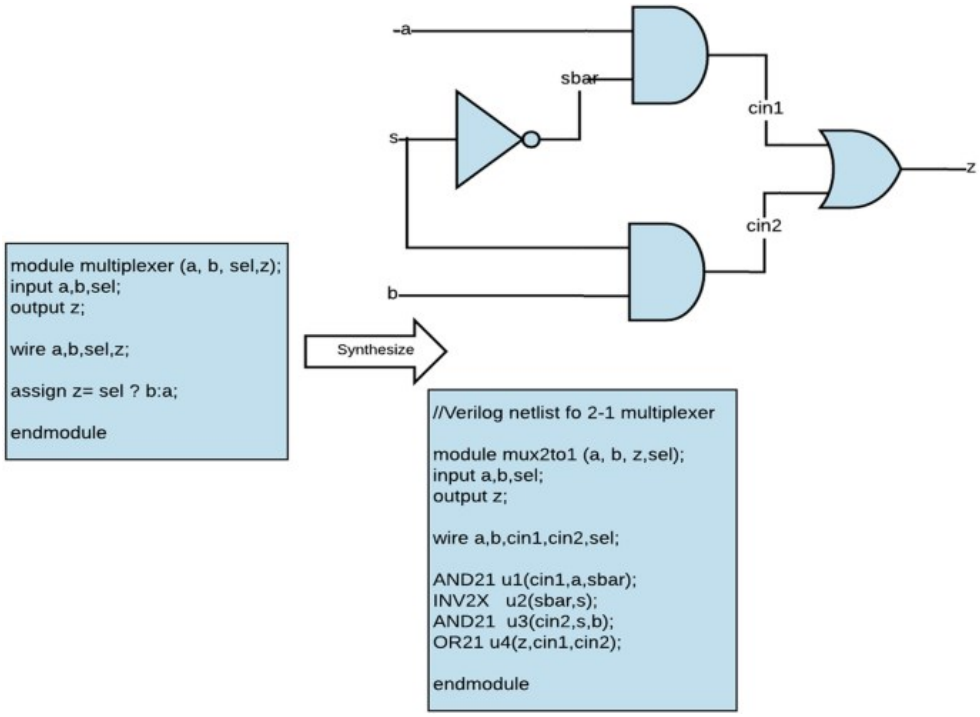


Figure 8. Logic Synthesis process [1]

To make packaging and reusing IPs easier, one can use the IP-Xact standard. The IP-Xact standard uses a basic extensible markup language (XML) format and methodology to describe and package IPs. IP-Xact standard is developed by Accellera. IP-Xact is developed to solve problems like integrating different IPs from different vendors and sources. Some key features of IP-Xact are metadata description, hierarchical structure, register definitions and bus interfaces. In a typical IP design work, usually some kind of tool is used build this XML format file describing the IP. An example of IP-Xact XML file is shown in appendix 1.

# 3 IP BLOCK IMPLEMENTATION

During my traineeship at Nokia, I had a great opportunity to put the theory into practice. My job was to design an IP block by following the tasks discussed in this thesis. Chapters 3.1 – 3.4 covers my experiences as a design engineer. Figure 9 illustrates the inputs and outputs of these design steps in practice.

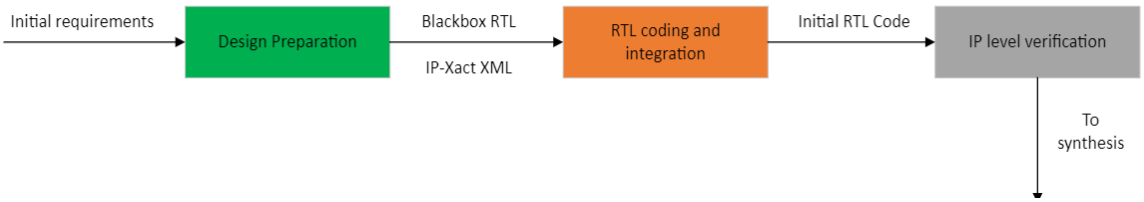


Figure 9. Design steps in practice

## 3.1 Design preparation

The design began with the design specification. I was given SoC-level information about the environment that the IP would be integrated in to. Based on that information, a basic block-level diagram was first drawn from which I could easily see the standard communication interfaces, as well as some standalone inputs and outputs used in the design. After I had the black-box diagram ready, the integration kind of already started. I used an industry standard tool to create the IP-Xact XML description of the black-box design. Using this tool, I could also export a ready-made HDL file that contained all the standard communication interfaces as well as the other standalone signals. With this tool, I was able to parameterize signal widths so I could later change those if needed. The tool made all this information present in the XML file.

## 3.2 Internal Functionality Design, RTL coding and integration

Next step was to think about the internal functionality of the IP design. This was done by drawing the internal subunits inside the earlier drawn blackbox design. Different subunits were connected with lines to easily see how they are related to each other. In this step it was also crucial to take notes about each of the subunits.

I started the RTL coding after the internal functionality design. I began by first deciding which of these subunits I want to include in the same HDL file, and which subunits need their completely own file. For example, I wanted to include the standard communication interface functionality in its own file. During the coding phase, I used a linting tool to make sure my RTL code doesn't have any obvious errors or bugs. The linting tool also reported stylistic errors and gave a warning if it noticed some suspicious constructs. The tool I used also let me include multiple files to its environment so that it could analyze the design hierarchy. It turned out to be extremely useful since I accidentally made several mistakes in instantiating the modules together.

At this point I had the IP-Xact XML ready, as well as the HDL code. Before proceeding to the verification tasks, I was asked to integrate my IP as a part of a bigger entity that was under development in Nokia. The integration was done by using the same IP-Xact packaging tool as earlier. The integration was done by opening the bigger entity in the IP-Xact packaging tool

and then using the tool to include the information about my IP to the whole Nokia project's IP-Xact file.

### 3.3 IP level verification

The verification phase took part after the integration. The verification plan was to use formal verification to find possible bugs from the communication interfaces or from the internal functionality. The verification process started with getting familiar with AIPs. Using these AIPs made verifying the communication interface faster, as I didn't need to write hundreds of assertions for this already known standard interface. The idea was to connect these AIPs to my newly designed IP, so that a formal verification tool could read it and perform the actual formal verification. Before I could run the verification, the tool itself needed a script file which contained a sequence of commands and instructions that were executed by the formal verification tool. The formal verification tool alerted that some of the assertions didn't pass, which meant the design didn't work as intended. The HDL code was modified accordingly to meet the requirements and pass the assertions. As discussed earlier, some IP design tasks happen in parallel. This was a perfect example of such a case. While writing this thesis, the verification is still in progress. It just emphasizes how the verification takes the most amount of work in high-tech development.

### 3.4 Thoughts about working in high-tech development

While working with the design project at Nokia, I quickly noticed that even though the IP design tasks have similarities regardless of the project or the company, every company has their own licenses with EDA vendors. For this reason, the user guides for tools are not available for public use. This means that there is no information available for these tools by just simply googling. This is why getting familiar with reading the official user guides and EDA software documentation is a must have skill to be able to work in a high-tech company.

# 4 DISCUSSION

Every IP design task plays its own important unique role. While working with my design at Nokia, I quickly noticed how the order of tasks is somewhat fixed, but some of the tasks can and will happen in parallel. In my design work, the verification and HDL coding happened somewhat parallel. It was also noticed that the integration part could take place after my IP design had its inputs and outputs ready. That way other teams could see my IP on layouts, for example. During the tasks, I noticed that the verification part was the one being the most challenging and it took the longest. I also quickly noticed how important it is to get familiar with reading the official user guides and manuals.

It was glad to see that the theory and practical work at Nokia supported each other well. Many things discussed in the theory part got a reasonable explanation during the practical work, for example the use of linting tools.

# 5 SUMMARY

In this thesis, the purpose was to examine what kind of steps and tasks are included in designing an IP block in a high-tech company. The thesis began with a theory part on general IP design tasks. As this thesis was done as a part of my internship at Nokia, the following chapters contained experiences about how to put this theory in practice. During this thesis it was found that designing an IP block involves several additional minor steps in addition to the general ones introduced in the theory chapter.

# 6 REFERENCES


[1] Chakravarthi, V. S. (2022). A practical approach to VLSI System on Chip (SoC) design: A comprehensive guide (Second edition.). Springer International Publishing.

[2] AMBA AXI4 and AXI4-Lite specification: https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview

[3] IEEE. (2017) IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017

[4] SoC Verification Flow and Methodologies URL: https://www.design-reuse.com/articles/52508/soc-verification-flow-and-methodologies.html

[5] Meyer, A. (2003). Principles of functional verification. Newnes.

[6] SystemVerilog Asserions Basics https://www.systemverilog.io/verification/sva-basics/

[7] SystemVerilog Assertions (ChipVerify) https://www.chipverify.com/systemverilog/systemverilog-assertions

[8] IP-Xact standard URL: https://www.accellera.org/downloads/standards/ip-xact

[9] Introduction to AMBA AXI4 https://developer.arm.com//media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/102202_0100_01_Introduction_to_AMBA_AXI.pdf?revision=369ad681-f926-47b0-81be-42813d39e132

# 7 APPENDICES

Appendix 1 IP-Xact XML file example
Appendix 2 List of AXI4 signals

Appendix 1 IP-Xact XML file example

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009"
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009/index.xsd">
  <spirit:vendor>Vendor</spirit:vendor>
  <spirit:library>Library</spirit:library>
  <spirit:name>alu</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:model>
    <spirit:ports>
      <spirit:port>
        <spirit:name>integer_operand_1</spirit:name>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
          <spirit:vector>
            <spirit:left>7</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:name>integer_operand_2</spirit:name>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
          <spirit:vector>
            <spirit:left>7</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:name>opcode</spirit:name>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
          <spirit:vector>
            <spirit:left>3</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:name>integer_result</spirit:name>
        <spirit:wire>
          <spirit:direction>out</spirit:direction>
          <spirit:vector>
            <spirit:left>7</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
    </spirit:ports>
  </spirit:model>
</spirit:component>
```

Appendix 2 List of AXI4 signals

| Global | Write address channel | Write data channel | Write response channel | Read address channel | Read data channel |
|---|---|---|---|---|---|
| ACLK | AWVALID | WVALID | BVALID | ARVALID | RVALID |
| ARESETn | AWREADY | WREADY | BREADY | ARREADY | RREADY |
| | AWID | WID | BID | ARID | RID |
| | AWADDR | WDATA | BRESP | ARADDR | RDATA |
| | AWLEN | WSTRB | BUSER | ARLEN | RRESP |
| | AWSIZE | WLAST | | ARSIZE | RLAST |
| | AWBURST | WUSER | | ARBURST | RUSER |
| | AWLOCK | | | ARCLOCK | |
| | AWCACHE | | | ARCACHE | |
| | AWPROT | | | ARPROT | |
| | AWQOS | | | ARQOS | |
| | AWREGION | | | ARREGION | |
| | AWUSER | | | ARUSER | |