Customized Processor Design for 5G Data Link Layer Processing

LUKAS FORSBERG
PATRIC WARGEUS
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Customized Processor Design for 5G Data Link Layer Processing

Lukas Forsberg
elt15lfo@student.lu.se
Patric Wargeus
elt14pwa@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Joachim Rodrigues

Examiner: Erik Larsson

June 28, 2020

Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgements

We would like to thank the staff at Huawei Technologies Sweden AB in Lund for their help, especially our supervisors Johan Hokfelt and Daniel Hedberg.

We would also like to thank Zdenek Prikryl and staff at Codasip for their quick and helpful responses to the many questions that came up throughout the thesis.

Contents

List of Figures

List of Tables

List of Code

Abstract

This thesis aims to explore the workflow related to designing an application specific instruction-set processor (ASIP). An ASIP is a processor similar to a hardware accelerator (HAC) in terms of performance and efficiency, but containing elements of general purpose processors (GPPs) when it comes to programmability and flexibility. The thesis centers around the design of an ASIP which will handle layer-2 processing in the 5G uplink i.e. keeping track of resources that are used by the user-equipment (UE) device. The ASIP and its related workflow are a relatively new concept in the wireless communications field; historically the large phone manufacturers have bought or licensed intellectual property (IP) from large chip designers such as ARM or Qualcomm, and then designed their applications around the framework that these chipsets provide. The main driving factor behind this exploration of the ASIP as a competitor to the GPP and the HAC is the relative maturity of design tools and the need for ever smaller devices, where efficiency in both power and size while keeping performance high is of utmost importance. Along with greater efficiency, today's devices are also often required to have some sort of design flexibility to facilitate changing standards or device usage cases.

The design tool chosen for use in this thesis is Codasip Studio, which has a workflow similar to other chipset design tools: a description of the architecture and it's instruction set architecture (ISA) is constructed, then after testing this behavioural representation it is sequentialized into the pipeline model and simulated. The final step is testing the firmware and peripherals on the simulated processor, before a VHDL or Verilog design is generated by the tool ready to export for register-transfer level (RTL) synthesis. The ASIP in this thesis is designed to run seven tasks which it switches between depending on what type of data processing is required or available at the moment. The design finalized in the thesis contains three tasks that are completely implemented and one task that is partially completed but not synthesized in RTL. The assumption that the remaining tasks have a similar complexity means that the results can be extrapolated to give an approximation of the entire processor. The total number of implemented instructions is 88. Of these 88 instructions, 55 are ASIs and 33 are part of the base instruction set, the set needed for the processor to be Turing complete, and therefore able to act as a GPP. The synthesized ASIP design is compared to several ARM equivalents in power consumption, area usage and instruction efficiency; the amount of instructions that are needed to complete the test firmware loop. The results prove

that the ASIP is a superior choice to the other processors, in this specific use case, by providing much higher throughput at roughly the same power consumption and area usage. In regards to the HAC comparison, no data was available to compare with in this specific case, so the comparison in this thesis is mostly a subjective one in regard to the design process.

Popular Science Summary

How do you design a processor today? When you think of the processor in your mobile phone, what tasks does it have to perform? Should it be able to run all your applications as well as handle network communication, or should it only run certain specific tasks, and how would the specific tasks impact the size and power consumed by the processor? There are many different approaches to this design problem, in previous generations of mobile networking standards, phone companies have often used general purpose processors (GPPs) for the network communication processing. A GPP is a processor designed to work sufficiently for a wide array of different applications, while generally not excelling at any single task. This is an inefficiency that becomes problematic in 5G, where performance and low power consumption are more important than ever. What if the company that designed the software instead decided to build a processor tailored exactly to the needs it had?

The flexibility of a processor is often defined by it's programmability i.e. how much can it's functionality be changed after it is in the final product. General purpose processors are generally the most flexible and dedicated hardware such as hardware accelerators (HAC) the least. Holding the middle ground between the two are application specific instruction-set processors (ASIPs); which are able to move closer to either of the former depending on the needs of the design. To design an ASIP efficiently there needs to be a clear idea of what kind of task it should perform, since the physical form of the processor will change depending on what it should do. When this is decided, a design tool is needed to create a model of the processor, which can then in turn be used by other tools to provide the final physical description of the processor. If the tool used is well designed it can significantly speed up the process and provide the user with useful information and means to test the design without having to physically construct it. One such tool is Codasip Studio, and when it is used together with it's own language to describe an ASIP; it gives area and power consumption results very similar to current advanced GPPs with the added benefit of the ASIP being much more efficient at performing the specific task it is designed for. If a similar tool was used to design an HAC for the same application, the performance results might be a bit better, but it's behaviour would also not be able to be changed after the fact. That is the most compelling thing about ASIPs, they can be designed to be exactly as flexible as needed so no part of the processor is wasted or superfluous. Design tools such

as Codasip Studio enable simplifications in the development process which make it faster and easier to use than traditional development methods.

Chapter 1

Introduction

1.1 Motivation

The introduction of 5G is said to mark a new milestone for the future of mobile networking and brings significant improvements over previous generations through increased bandwidth, reduced latency and increased reliability [1]. Increased network performance will pave the way for new technologies and innovations such as self-driving cars [2], Industry 4.0 [3], and the Internet of things (IoT) [4]. However, this comes with a price, due to the end of Moore's law and Dennard scaling, the improvement in performance and power consumption has stagnated on today's general purpose processors (GPP) [5]. As a result, the increased data transfer rate that 5G brings will become increasingly difficult for a GPP to handle efficiently without increasing power consumption and chip area. This becomes an issue especially for for mobile phones and IoT devices, which have very strict limitations on power availability and size. One solution to the problem for many developers has been to rely on specialised hardware for demanding data processing, however this also causes problems as the flexibility of hardware implementations is very low. In order to develop hardware without the risk of it being already outdated at release, and for it to able to handle high data processing rates with low power consumption, a different approach is required. Application specific instruction-set processors (ASIPs) running specialised instruction set architectures (ISA) are one such approach. ASIPs have historically held the middle ground between GPPs and dedicated hardware such as hardware accelerators (HAC), in that they combine the flexibility of GPPs with some of the speed of the HAC.

Huawei is a major player on the wireless market, accounting for roughly 15% of sold mobile telephones in 2019 [6]. Huawei is investing heavily in 5G development, both in the network and user plane, and are posed to become industry leading when 5G becomes the standard. Huawei is well aware of the problems that the combination of extreme data rates and ever shrinking device size bring, and is therefore developing and testing new solutions. Previously much of the network functionality in their devices was performed on hardware accelerators (HACs) and GPPs integrated on ASICs. Due to the previously mentioned low flexibility of HACs and low performance of GPPs, it is of interest to investigate if parts of the data processing can be implemented on an ASIP. The idea is to maintain the performance of dedicated hardware while maintaining the flexibility of a GPP. The

ASIP design process is heavily reliant on developmental tools; due to this and the impact of the current political climate on suppliers, it is of also in Huawei's interest to investigate available alternatives in this domain. This thesis will therefore investigate whether it is possible to develop an ASIP with a new ASIP design tool that can handle the processing requirements of new 5G standards while still keeping some of the flexibility required to be updated to new 5G releases, as well as provide a first hand look at a new development process for ASIPs.

1.2 Purpose

The purpose of this thesis is to design an ASIP that handles uplink communication in the user-plane with the help of Codasip Studio [7]; an ASIP design tool software, and analyze both the results and the design process in comparison to implementing the application using other methods. The design process will be centered around implementing an ISA that can provide high throughput for 5G networking, as well as flexibility for future updates to the 5G standard and beyond. The ASIP is part of a system-on-chip (SoC) subsystem that handles header management of radio link control (RLC) and packet data convergence protocol (PDCP) information in layer 2 of the uplink according to the standard set by 3GPP in 5G release 15 [8]. The design of the ASIP will require the creation of an instruction accurate (IA) model and a cycle accurate (CA) model. The end goal of the thesis is to attempt to provide information on throughput, chip area, power consumption and ISA flexibility; through both simulations and register transfer level (RTL) synthesis, that can then be used to compare the ASIP with other hardware and GPP solutions. The thesis will also serve as a review of the Codasip Studio tool and describe in detail the ASIP workflow, where the processor is designed around the application.

1.3 Constraints

The design of a processor from scratch is a demanding project even for a team of engineers, therefore a few constraints will be imposed on the thesis to make it more manageable for two students. Huawei will provide the SoC design structure that the ASIP will fit into as well as pseudo-code for many of the functions that the ASIP is intended to perform. As for the processor design itself; very long instruction words (VLIW) might provide benefits to the flexibility of the ISA, but will also increase compiler complexity and is therefore beyond both time frame and scope of this thesis. The functionality of the ASIP aims for coarse grain programmability; the ISA, building blocks, functional units and specialised memories are only customized for a specific application and the ASIP is not meant to be used in any other circumstance. The ISA implementaion will therefore not contain redundant instructions that might be used in future, the sought after flexibility will instead be implemented in the instructions that are already proposed in the pseudo-code provided. The thesis will not focus greatly on compiler construction either; usually this is very important for testing the processor and firmware, but

in this case the expected firmware size will be small enough that it can be written in Assembly without any problems.

Chapter 2
Background

2.1 5G New-Radio

The technical work on 5G New-Radio (NR) began in spring 2016 as part of a study item in 3GPP release 14, the work was based on an initial kick-off workshop in 2015. Many technical solutions were studied and proposed in this early stage and due to the tight time schedule, some were appropriated already in this phase. The work continued throughout 2017 and became a standalone work item with it's first specification being released in conjunction with release 15, mostly to meet commercial pressure from early 5G developers. This thesis will be working with release 15, finalised in 2018, with it's late release frozen in mid-2019 according to 2.1. The later revision to release 15 offers standalone 5G access, without the need for LTE functionality in the network, but also includes backwards compatibility with the LTE standard. Standalone 5G provides many benefits, some of the most important are[9]:

- Use of higher frequency bands in a much wider spectrum; licensed spectrum from 1-52.6 GHz with support for unlicensed extension planned in release 16. This gives much better support for very wide transmission bandwidths and as a result much higher data rates.
- Simplified design to help with network energy performance and reduction of interference.
- Forward compatibility adds overhead and support for yet unknown use cases and technologies.
- Lower latency to improve performance and connection as well as enabling new use cases.
- A design focus facilitating and encouraging the use of beamforming and a massive number of antennas for both data transmission and control-plane procedures such as initial access.

These points can be seen as a brief overview of the motivation behind the development of 5G; the main points that apply to this thesis and the ASIP it proposes are found in the structural definitions of layer-2 processing. A general view of the user-plane and control-plane protocol stack can be seen in Figure

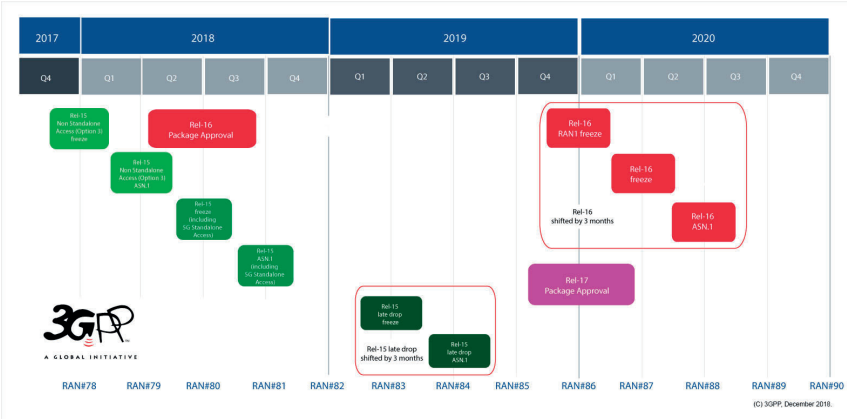

Figure 2.1: NR release schedule [8].

2.2; where UE denotes the user-equipment (UE) device and gNB, the g-node-b or network connection. Many of the protocol layers are similar to those seen in LTE or 4G; the main difference is in the use of the service data adaption protocol (SDAP), where a number of different quality-of-service (QoS) flows can be used to route IP-packets according to their specific QoS requirements. The SDAP is only active when the user is connected to the 5G core network i.e. when operating in 5G standalone mode. The main focus of the ASIP designed in this thesis is handling PDCP and RLC headers in the 5G uplink, a data flow chart of which can be seen in Figure 2.3. After the digital user-plane layers come the logical layers before antenna transmission; medium-acess control (MAC) handles multiplexing of logical channels, hybrid-ARQ transmissions, scheduling and scheduling functions. MAC is an older but widely used protocol that has had it's header structure changed in NR to facilitate lower latency transmissions than in LTE, the revised header structure can be seen in Figure 2.4. MAC provides functionality to the RLC layer in the form of logical channels; either control channels used for transmission of control and configuration information or traffic channels used for user data. The outermost layer of the user-plane is the physical layer (PHY), where coding/decoding and modulation/demodulation, multi-antenna mapping as well as mapping of logical channels to specific time and frequency resources (physical channels) is done. This brief overview of the user-plane protocol stack is followed by a more detailed overview of the protocols relevant in this thesis, PDCP and RLC. The detailed overview will give some indication about which parts of the protocol processing is performed by the ASIP.

2.1.1 Overview of the PDCP and RLC

The following descriptions are taken from 3GPP's series 38 of specifications[8], so to avoid duplicating the information, the focus of this section is to give an overview of what functions the ASIP designed in this thesis performs. The PDCP performs many functions, the main being header compression to reduce the total number of transmission bits, and can operate together with the RLC in both

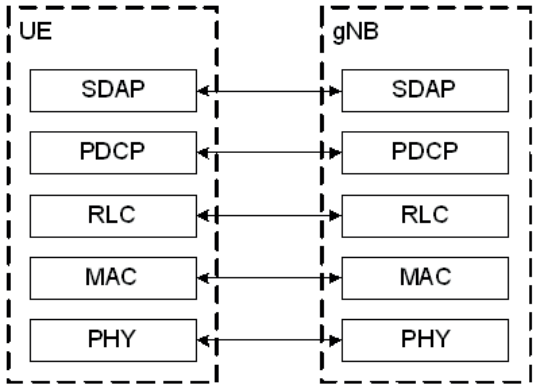

Figure 2.2: User and control-plane protocol stack[8].

unacknowledged (UM) and acknowledged (AM) mode. The header compression is done using the robust header compression (ROHC) framework, which is a set of standardised compression algorithms widely used in mobile communications. Apart from header compression the PDCP is also responsible for ciphering and eavesdropping protection in both user and control planes, both to protect end-user data and to ensure that control messages are accurate and from the correct source. The PDCP is also responsible for reversing these operations at the receiver side, i.e. deciphering and decompressing. The PDCP layer is configured by upper layers and each PDCP entity can be mapped to one, two or four RLC entities and each UE device can have several different PDCP entities; each of which carries the data of one radio bearer. The ASIP does not perform all of these tasks, for each PDCP entity it will:

- Inform upper layers to set up relevant data structures when a new PDCP entity is established.
- Keep track of the transmisson/reception status, which is the window state for the PDCP entity i.e. when a certain entity is allowed to send or receive.
- Keep track of resend and suspend operations for each PDCP entity and inform upper layers.
- Increment and decrement the sequence number (SN) of each PDCP entity to keep track of when they can be discarded.
- Store number of memory references to each entity and inform upper layers to discard data if it is no longer referenced and is outside the allowed sequence number.
- Handle ACK and NACK information received from RLC and inform upper layers of which PDCP entities are affected.

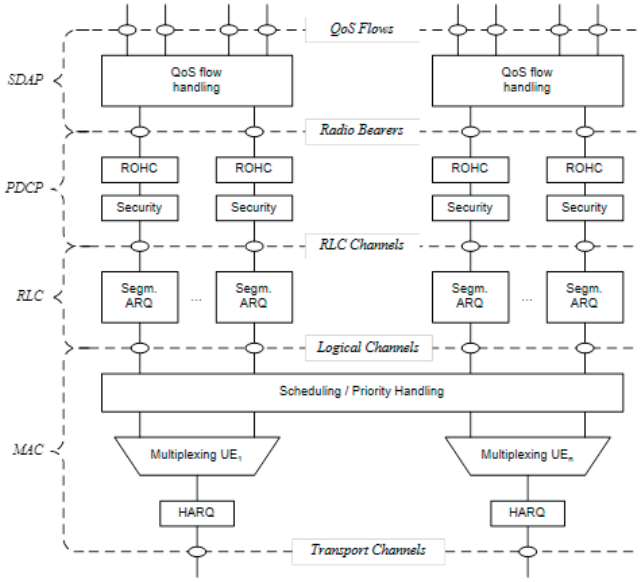

Figure 2.3: Layer-2 protocol stack for 5G uplink [8].

Servicing the PDCP is the RLC, a lower level protocol which is closer to the send/receive physical channels than the PDCP. The RLC mainly handles the transfer of upper layer packet data units (PDUs), error correction through ARQ and all operations related to RLC service data units (SDUs), as well as RLC establishment and re-establishment if connection is lost. Since several RLC entities can be associated with each PDCP entity, the RLC layer acts as a connection handler for the PDCP layer; it keeps track of where and when data can be sent on each radio bearer associated with a specific RLC entity. The functions performed by the ASIP on the RLC entities are very similar to the PDCP case, the only real difference being the different data structure and the fact that the RLC entity has it's own SN and performs similar operations on it's own variables.

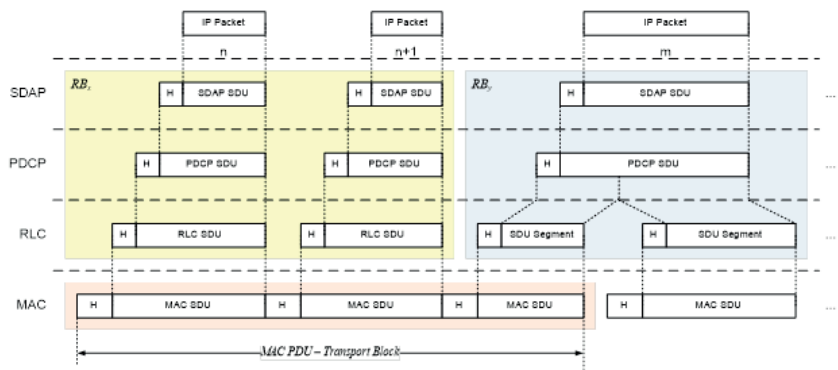

Figure 2.4: Header structure for 5G NR release 15 [8].

2.2 Processing Architecture

2.2.1 General Purpose Processors

A general purpose processor is exactly what the name indicates, general. It is designed to work with sufficient performance at a very high variety of applications. General purpose processors are designed to score high in terms of performance in general benchmark suits such as the SPEC benchmark, which test the performance for a mixed range of applications. Embedded systems on the other hand, are often developed to do a very specific task, therefore a good performance score in a diverse range of applications is not necessarily a good thing. The primary aim of an embedded system is to satisfy the design goals for the problem at hand, with general purpose computing being less important. [10]

General purpose processors are often equipped with hardware optimizations, a few examples are:

- Memory caches to reduce memory access time.
- Branch prediction schemes to reduce control hazards.
- Scoreboard/Tomasulo algorithm implementations to prevent data hazards.

These optimizations require a lot extra of hardware resources such as registers and combinational logic which in non-general purpose applications could be unused. If the order, timing and reuse of arriving data is known, there is no need for caches. In applications where the program flow is already known at system construction, there is no need for branch predictions or Scoreboard/Tomasulo algorithm. Therefore the total area and power consumption could be significantly reduced by removing these optimizations. The total chip area and it's power consumption is extremely important for modems supporting 5G-NR, due to the limited energy and area resources available to small devices. The modems are also often planned to be produced in large numbers, thus it is important to make them as cheap as possible to produce while still meeting power and area constraints. Usually when hardware manufacturing companies are faced with the problem that a general purpose processor is not sufficient for a specific task, they go with an HAC solution instead. HAC designs do however come with their own limitations, as described in next section.

2.2.2 Hardware accelerators

Every new mobile generation has introduced higher transmission speeds, and this together with the ever shrinking device and transistor size has made it increasingly difficult to construct and maintain design tools which can keep up with the development. The number of transistors on a chip increases linearly together with design complexity and this leads to a disparity between chip complexity and development time if current design tools do not allow the increase in productivity to follow the transistor amount curve. This disparity between the transistor amount used in modern very large scale integration (VLSI) chips and the growth rate in design productivity (number of transistors/staff-month) has increased heavily the

last 20 years. The problem has been called "The Design Productivity Crisis" and it is forcing VLSI chip manufacturers to increase the development time for each new generation of hardware. To overcome this difficulty, chip manufactures are currently focusing on finding ways to make the abstraction level of the design process as high as possible and also to make the design reuse rate as high as possible [11]. The limiting factor in today's digital hardware development is therefore more in the design tool and it's productivity limitations, and not inherently a part of the HAC architecture, as the increasing device complexity has made older tools and hardware description languages (HDL) too slow for many design processes and their time constraints. Many developers are therefore choosing to resort to software solutions instead as developing and debugging these solutions is often many times cheaper and quicker than developing and debugging a hardware solution.

Another limiting factor of HAC design is that the construction of silicon mask set for hardware is becoming more expensive to manufacture. Decreased transistor size will results in increased material cost and higher failure rates. The programmable nature of an ASIP or a GPP enables manufacturing of larger volumes at less risk due to the fact that similar applications can be mapped to the same hardware and can be used for different generations of products to a larger extent than an HAC. In addition programmable solutions also provides a lower risk in case of firmware and hardware flaws, due to their programmability, and often a shorter time to market than pure hardware designs. This is important in a competitive field, such as the release of 5G-NR, where different actors are competing against each other to deliver a marketable product as soon as possible that defines the 5G standard.

2.2.3 Application Specific Instruction-Set Processor

ASIP design can generally be described as the creation of a new processor, where the instruction set and architecture are customized for a targeted set of applications. The design goal for an ASIP is to be more efficient then a GPP in terms of area and power consumption and at the same time have similar performance as an HAC. Some reductions in design quality are inevitable in terms of area, delay and power when comparing a ASIP to an HAC, but this is weighted against the productivity benefits of software solutions [12]. Typical ASIPs are based on a simple reduced instruction set computer (RISC) architecture which allows the ASIPs ISA to be targeted by a standard compiler. The ISA is then enhanced with dedicated instructions and special application specific registers to increase performance for the target application. As pictured in Figure 2.5 a major design consideration when constructing an ASIP is the relationship between efficiency and flexibility. It is important to decide early in the design project where in Figure 2.5 the ASIP should belong. In this project efficiency is valued highest due to the requirements placed upon it by a market compatible 5G-NR modem; high performance, low power and low production cost. These requirements place this project's ASIP in the upper left of Figure 2.5 marked by "Project ASIP".

2.2.4 Design Tools for ASIP design

Using a HDL such as VHDL or Verilog to design the processor described in this thesis would be a very complex and time consuming task and would most likely result in a high development cost. The processor is usually the most advanced piece of hardware on a chip and relying on low-level VHDL and Verilog design tools would most likely not be sustainable for companies like Huawei, who are constantly under pressure to get products to market as quickly as possible. To make the ASIP design process feasible, programming languages specifically used for high level processor description called architecture description languages (ADL) has emerged within the VLSI industry. These ADL's can be seen as a high level HDL and relieves the hardware developer of implementation responsibility which are then passed on a ASIP design tool. The ASIP design tool then parse the ADL code and generates a RTL of the target processor.

An ADL is not only used to simplify the hardware development process, but is also used to construct a compiler targeting the processor's ISA. To take serious advantage the productivity benefits of the software solutions, a compiler for some programming language has to be constructed for the ASIP. Software written in assembly tends to be far more complex to develop and debug which in turn decreases productivity compared to writing code in a high level language. Therefore an efficient compiler for languages such as C or C++ becomes more important as the code size grows. Constructing a custom compiler targeting an application specific ISA can be very complex due to the wide range of instructions that are normally contained within the ISA. However, a compiler is necessary to reach a sufficient ASIP design productivity rate due to the ease of use of being able to target the processors ISA from a higher abstraction level. To summarise, the purpose of an ADL is to be able to generate a RTL of the target processor from a single model description along with it's compiler, cycle accurate and instruction accurate instruction set simulators, a testbench and other verification tools.

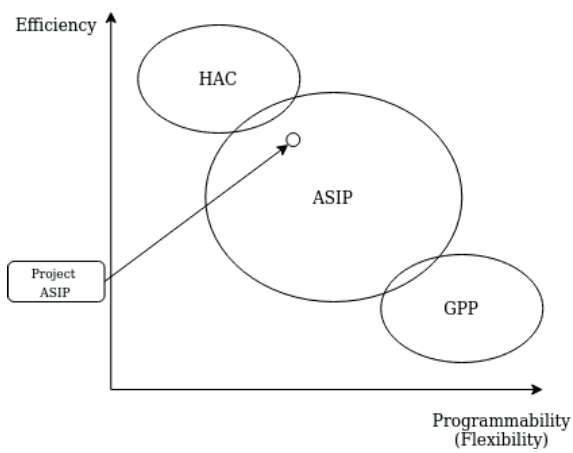

Figure 2.5: ASIP vs GPP and HAC [12].

In recent years there has emerged a couple of ASIP design tools which try to automate and simplify as much of the ASIP design process as possible. The current tool dominating the market is Synopsys' "ASIP Designer" but the smaller competitor Codasip "Codasip Studio" has increased in popularity during the last couple of years. Both design tools use their own ADL for the ASIP's model description. The tools provide a interface where the designer can easily describe the ISA, the pipeline stages and the compiler. The tools then generate, based on the description provided, a RTL description of the processor and a C/C++ compiler. These kinds of tools are essential for an efficient and sustainable ASIP design process. A graphical representation of the capabilities of a typical ADL can be seen in Figure 2.6.

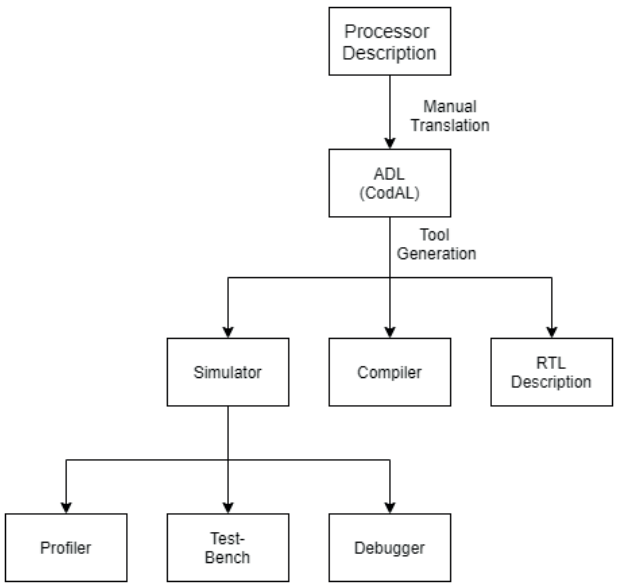

Figure 2.6: Role of ADLs

2.2.5 Codasip

Codasip Studio is an ASIP design tool developed by the relatively new startup company Codasip. The tool aims for rapid and automated ASIP design methodology while simultaneously generating high performance register-transfer level (RTL) descriptions together with a C/C++ compiler for the processor. The processors hardware and functionality is described in the tool's own C-like ADL called CodAL. The design process in Codasip Studio is divided into two main stages, the instruction accurate model (IA) and the cycle accurate model (CA). The IA model is used to describe the initial processor instruction set on a functional level without

any microarchitectural details. From this high level description, the tool can then generate a C/C++ compiler, assembler and simulator for testing and debugging. When the IA model is implemented and tested, the designers can proceed with the CA model which includes the microarchitectural details such as pipelining, timing and signaling. The tool can then generate a simulator for the microarchitectural description as well as RTL hardware descriptions in either VHDL, Verilog or SystemVerilog. The ASIP design model is divided into three parts in the studio environment, the IA, CA and shared model resources. A flowchart of Codasip's design flow can be seen in Figure 2.7.

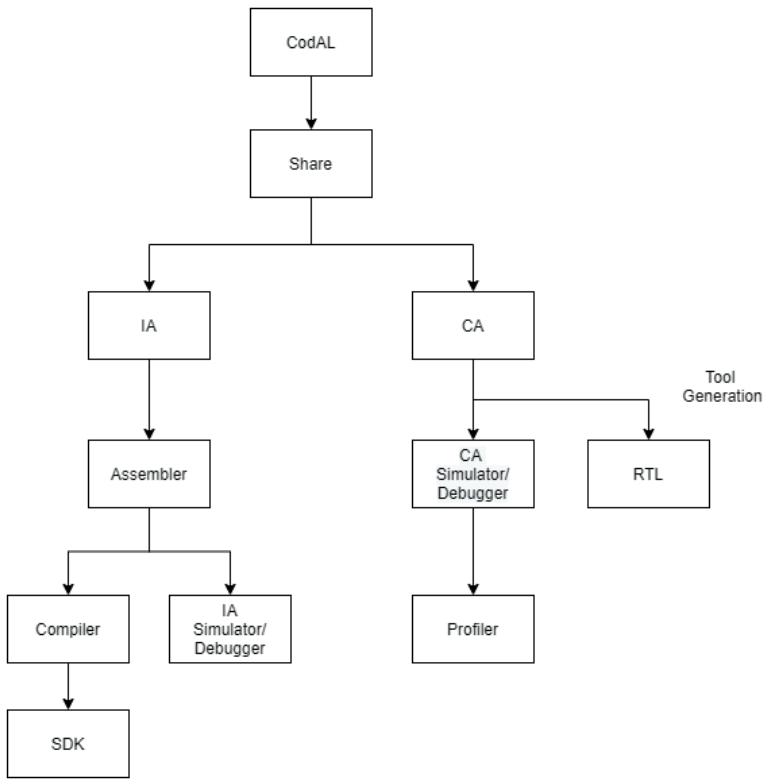

Figure 2.7: Codasip flowchart

Chapter 3
Method

This thesis will involve the design of a processor using the Codasip Studio ASIP design tool. Which means that Codasip's own ADL, CodAL will be used to design and simulate the processor. In addition a C compiler targeting the ASIP's base ISA described in Appendix A.1 will be constructed and generated with the help of the Codasip Studio environment. The firmware will be written in assembly. The SoC environment in which the ASIP will operate will be simulated to test the ASIP in Codasip Studio by describing the functionality and behaviour of the external peripherals in C++ .

3.1 Design Guidelines

As described previously, the ASIP will be designed to closer resemble an HAC than a GPP. This means that the ASIP will be equipped with a large set of application specific instructions (ASI). These ASIs will implemented mostly as static, non-configurable functional units with low flexibility and high performance. The processor will be synthesized on a 7 nm transistor technology, and is expected to run at a frequency of 400 MHz or lower. This implies that pipeline stages could be designed relatively deep; in other words signals are expected to be able to pass through a large amount of digital logic in the time span of one clock cycle. A synthesis report of the generated RTL will be done once the processor is finished.

3.2 Project Phases

3.2.1 Instruction Accurate Model

The instruction accurate model phase is the first phase in the ASIP design process and is where the ASIs are defined. For each instruction the assembly syntax and the binary representation have to be described in the CodAL model. Once complete an assembler and disassembler targeting the ASIP's ISA can be generated by the tool. In addition the functional aspect of the instructions can be described in a software manner to allow for early stage simulation and prototyping of the ISA without the microachitecture being defined. Only the hardware that is relevant for the functionality for specific instructions has to be defined in the IA model,

such as general purpose registers and application specific registers used by the instructions.

Once the assembly syntax, binary representation and functionality is described for each instruction, it is possible to generate a C or a C++ compiler targeting the custom ISA. This step requires the addition of directives for the compiler on how to interpret certain instructions. The application specific instructions to be implemented in this project are not supposed to be used by the compiler, but rather it is the programmer's responsibility to call them directly from the C code with intrinsic function calls. The ASIP's firmware will therefore mostly be a C program calling intrinsic functions which simply tells the compiler to put an application specific instruction at a specific code line. As opposed to when the compiler schedules a ASI based upon a set of expressions in C Therefore the construction of the compiler will be rather simple and was given low priority in this thesis.

3.2.2 Testing of IA model

The second phase involves testing the ASIs described in the first phase to guarantee that they are aligned with the application's functional specification. The tool provides a framework to unit test the application specific instructions. The framework allows the tester to define the state of architectural registers for each clock cycle the test is running. Python scripts can also be written to check for error codes in the return registers once the test has finished execution. In this manner automated test suites can be designed in an effective manner.

In this project the existing testing framework was not used extensively. This was due to the fact that the framework requires an existing and working C or C++ compiler generated within the tool. The generation of a functioning compiler was determined to not fit within the thesis time plan, and since the firmware is to be written in Assembly, it was not considered crucial to the project. The IA testing methodology used in this project was instead designed as follows; two test instructions, "test_start" and "test_end" which both take the instruction to be tested as argument. The "test_start" sets up the initial state of the application specific registers used by the instruction to be tested. The "test_end" is used after the tested instruction has executed and checks so it manipulated the instruction specific registers correctly. This solution allowed for each instruction to be tested in isolation which simplifies debugging. The error checking was done with the simulator's builtin functions used for debugging, such as the typical assert function, which takes a boolean expression and a specified error message as argument. The function terminates the simulation if the expression is false and displays the error message. These instructions where omitted at synthesis.

3.2.3 Cycle Accurate Model

In the cycle accurate (CA) phase the microarchitecture and the pipeline stages are described and implemented. The CA model is developed in what could be seen as a high level HDL. Unlike the IA model, microarchitectural resources such as signals, external ports, bus interfaces to data/instruction memory, functional

units and the pipeline stages' specific resources have to be described. After the CA model is complete a CA simulator can be generated to test the entire processor implementation. It is from this CA model that the RTL description is generated and analyzed. The tool's profiler provides some functionality to compare different CA model implementation in terms of area and power usage but it can't give an exact figure in real standard units.

3.2.4 Testing of CA model

The ASIP's functionality is based around routing and processing incoming data from the SoC environment. Therefore to test the ASIP functionality in a cycle accurate manner, a cycle accurate SoC environment simulation has to be created as well. The tool provides different approaches for simulation of external components. The first approach is to simply feed the ASIP simulator with a text file which contains the input values of the ASIP's ports at specific clock cycles. The second approach is to link together the ASIP simulator with a SystemC model. The third is to use the tool's builtin support for describing external components in C++ . The latter was chosen for this thesis because the dynamic behaviour of the SoC did not fit the text file alternative. Furthermore the C++ behavioural description, in this specific case, is the easiest to connect to the ASIP simulator because the framework is provided with the tool. In the SystemC case some extra wrappers have to be constructed to link together the ASIP and the SystemC model.

3.2.5 Optimization

Once the CA model is tested and the ASIP works as tended, optimization can be done on the model description. The optimizations performed on the ASIP where mainly:

- Group together similar instructions so they can take advantage of the same functional unit and decoder to increase resource sharing.
- Replace complicated arithmetical operations with simpler bitwise operations to save hardware.
- Reduce the total amount of registers by letting tasks share resources if there is no risk for resource hazards.
- Limit the number of registers used by ASIs.

3.2.6 C Compiler Generation

The C compiler was constructed and finalised at the end of the project. The ASIs are not supposed to be used by the compiler unless they are called by their corresponding intrinsic function from a C program. The ASIs in this project perform very specific data processing and due to the performance requirements of the ASIP it has to be guaranteed from a software perspective the ASI are called as intended. Therefore it is logical to put the responsibility for the ASI call on the programmer and not on the compiler. Directives to the compiler was added in the IA model to simply ignore each ASI and generate a header file with the definitions of the intrinsic functions. Codasip Studio has the capability to generate a compiler which can analyse normal C code and determine if an expression could be translated into a custom instruction. This was done for the Base ISA described in Appendix A.1. The functionality of an instruction has to be defined in the IA model and the studio can determine from its functional description what the instructions does and the C expression it corresponds to, an example is showed in code 1. It is required to specify information about which registers the compiler can use and the registers' intended use. For example the program counter register and the register file which holds the general purpose registers. Registers that store the stack pointer, base pointer, return address, function results and function parameters have to be specified as well.

```
element i_add
{
    // The opcode for an instruction
    use opc_add as opc;

    // Define number of general purpose register used
    use reg_gpr as gpr_src2, gpr_src1, gpr_dst;

    // Assembly representation
    assembly { gpr_dst "= ADD " gpr_src1 "," gpr_src2 };

    // Binary representation
    binary { opc gpr_src1 gpr_dst gpr_src2 };

    // Provide compiler with information of the instruction
    semantics
    {
        uint32 src1, src2, result;

        // Read two gpr indexes from instruction argument
        src1 = rf_gpr[gpr_src1];
        src2 = rf_gpr[gpr_src2];

        // Two gprs are added together
        result = src1 + src2;

        // The result of the addition is stored in a gpr
        rf_gpr[gpr_dst] = result;
        /*
        * The compiler now knows this instruction
        * can be used to add two values together
        * and store the result.
        * The compiler can therefore map the C statement
        * a = b + c;
        * to this instruction.
        */
    };
};
```

Code 1: Compiler example.

3.3 Incremental Development

In the beginning of the project a waterfall design process model was considered where all steps described above would be done after each other in one iteration. Due to the risk of not completing the entire project in time and hence miss out on essential parts of the ASIP design process, an agile development methodology called the incremental build model was chosen instead. With an incremental build model the ASIP is designed, implemented and tested incrementally and a little more is added each time [13]. This implies that even if the entire processor would not be finished according to the original specifications in time , there would still exist a working processor with a sub part of the functionality implemented. The project was divided into two iterations. In the first iteration the base processor and ASIs for the two most simple tasks were constructed. In the second iteration ASIs for two more complex task were implemented.

3.4 Workflow Example

This section aims to give a brief overview of the workflow that this thesis aims to test, and that this project will be performed according to. The starting point for this project is a design document that has been prepared with regards to the standards document that is written by 3GPP with every new release of a 5G standard. An excerpt from 3GPP release 38 can be seen in Figure 3.1. The standards provided by 3GPP can be seen as a framework that all parties must abide by, with some leeway in terms of UE implementation decisions. The example in Figure 3.1 is from the PDCP uplink behavioural definition and describes how long the system should keep track of PDCP SDUs and when they should be discarded.

At reception of a PDCP SDU from upper layers, the transmitting PDCP entity shall:
- start the discardTimer associated with this PDCP SDU (if configured)
For a PDCP SDU received from upper layers, the transmitting PDCP entity shall:
- associate the COUNT value corresponding to TX_NEXT to this PDCP SDU;

NOTE 1: Associating more than half of the PDCP SN space of contiguous PDCP SDUs with PDCP SNs, when e.g., the PDCP SDU's are discarded or transmitted without acknowledgement, may cause HFN desynchronization problem. How to prevent HFN desynchronization problem is left up to UE implementation.

Figure 3.1: Original 3GPP behavioural description[8].

From the behavioural description the system designer can make decisions based on the implementation and internal structure of the UE device. In this case, the PDCP SDUs are received from a higher level software interface that also acts as a bridge to system memory. The ASIP should in this case check the associated reference counter (RC) and sequence number (SN) of a particular PDCP entity and release the related memory if the entity is not used and can be discarded according to the 3GPP definition of how long an entity should be saved. The pseudo code in Code 2 is the first step in the implementation process, and can be used together with the 3GPP description of PDCP entities to decide how the UE device should keep track of them.

```
1
2  Loop (COUNT){
3    Read current RC related to the SN of the PDCP entity
4    Decrease RC value
5    If ((RC after substract operation == 0) AND (SN ==
         RC_Release_Next)){
6      Read sdu address (PDCP func(Entity_Id, Count))[
7      Release memory
8      RC_Release_Next++
9      Return MEM_RC column memory based on RC_Release_Next
           judgement
10     Return PDCP_TX_REC column memory based on
           RC_Release_Next judgement
11   }
12   Else{
13     Write updated RC value to MEM_RC
14   }
15 }
```

Code 2: Pseudo-code based on description in figure 3.1 and system model.

The next part in the process is not mandatory, but makes behavioural analysis and later stages easier. Here, the pseudo code and entity descriptions are translated into C-code that is executable; which can then be used to confirm that the code performs according to expectations, as well as find complex instruction sequences that can be turned into ASIs. In Code 3 the instruction sequences that are within ASI brackets are highlighted as potential ASI candidates and also contain a call to a function that increases a cycle counter; to give a rough estimate of the cycle consumption of the program that will run on the ASIP. The C-code also works as a starting point for the firmware description of the ASIP, as it is essentially a behavioural description. Worth mentioning is that the C-code in code segment 3 is not a complete representation of Code 2, but rather a part of a larger task description.

```
1  while (NumRcToBeReleased > 0) {
2
3      ASI {MauLoadPdcpStates_Addr(PdcpSn,
           NumRcToBeReleased)}
4
5      for (int i=0; i< NumRcToBeReleased; i++) {
6
7          ASI {arAddr = addr.front(); addr.pop();}
8
9          ASI {MaaDeallocatePdcpSdu(arAddr);}
10
11         ASI {uint18 tmp = PdcpEntity[PEID].RcRelNext;
12                PdcpEntity.RcRelNext, PdcpEntity.WinMask;
13                PdcpEntity[PEID].RcRelNext, PdcpEntity.
                      WinMask;
14                RcReleaseFlag = RcReleaseFlag ||
15              ( ((PdcpEntity.RcRelNext & cRcRowIdxMask) ==
                     cRcRowIdxMask) &&
16              ((PdcpEntity.RcTxNext & ~cRcRowIdxMask) !=
                    tmp) );}
17     }
```

Code 3: C-code example with ASIs highlighted within brackets.

Before the instructions described in C-code can be translated into CodAL, the architecture of the processor needs to be described. The main architecture element in the IA model of the ASIP is the register description, and to be able to translate the instructions fully, a move away from variable manipulation to register value manipulation is required. Code 4 shows the architecture definition of a register in CodAL, where the size of the register in this case is the size of the PDCP entity that is to be stored in it.

```
1
2  register bit[PDCP_ENTITY_W] r_pdcp_entity{
3      reset = true;
4      default = 0;
5  };
```

Code 4: CodAL code example of a register definition.

To then translate the C-code from the previous part into functioning CodAL code, the instructions that are proposed as ASIs should be extracted and described. Code 5 shows how the final ASI instruction from the above code is described in CodAL, where the function "DEF_OPC" defines the opcode in three different ways. The instruction is then created as an element and is given an Assembly value, in this example just the opcode because it has no inherent inputs, and a

binary value. The instruction semantics section then describe the behaviour of the instruction in the IA model, where temporary variables are allowed to be used together with registers. In the last row of the code segment a call to read the databus is sent; in this case the read is described as a function, where it will later need to be described as a port read from an external component which is described in a C++ behavioural file.

```
1
2  DEF_OPC(SetReleaseFlag,"SetReleaseFlag",
       OPC_SetReleaseFlag)
3  element i_SetReleaseFlag {
4
5       use opc_SetReleaseFlag as opc;
6
7       assembly{opc};
8       binary{opc REMAINING_BITS(OPC_W)};
9
10 semantics {
11      uint32 temp;
12
13      temp = (r_pdcp_entity[PDCP_ENT_RC_REL_NEXT] + 1)
14      & r_pdcp_entity[PDCP_ENT_WINMASK];
15
16      r_pdcp_memref_entity = r_pdcp_entity[169..138] ::
17                             (uint18)temp :: r_pdcp_entity
                                   [121..0];
18
19      r_rc_release_flag = read_databus(GetReleaseFlag);
20      };
21 };
```

Code 5: CodAL code example of an instruction.

After the IA model architecture and instruction descriptions are complete and tested, they can then be turned into a CA model by sequentializing the instructions into a pipeline description along with a pipeline architecture model and instruction decoders. To translate the instruction in the previous code segment into a CA model it is divided into it's pipeline parts, of which only two out of four are shown in 6. In the execute stage (EX) the value of the opcode passed on from the previous decode stage is compared against a table of values and if it matches one, the instruction is completed in the same cycle. This means that all variable are described as intermediate signals between registers, and that reads and writes to registers are done separately. Here it can also be seen that the function calls to produce a value from an external component are replaced by actual value writes to a port. After the instruction is executed the writeback stage (WB) is where the values are finalized, read and stored in their respective registers.

```
1  event ex : pipeline(pipe.EX)
2  {
3      semantics{
4          switch(r_ex_opcode){
5
6              case OPC_SetReleaseFlag:
7
8              if(!bus_is_busy){
9
10             p_CmdStrobe = 1;
11             p_CmdOpcode = GetReleaseFlag;
12
13             r_pdcp_entity = r_pdcp_entity[169..138] ::
14                     (uint18) ~((r_pdcp_entity[
                           PDCP_ENT_RC_REL_NEXT] + 1) &
15                     r_pdcp_entity[PDCP_ENT_WINMASK]) ::
16                     r_pdcp_entity[121..0];
17                 };
18          break;
19         };
20     }
21 }
22
23 event wb : pipeline(pipe.WB)
24 {
25     semantics{
26         switch(r_wb_opcode){
27                 case OPC_SetReleaseFlag:
28                     r_rc_release_flag = p_ReadData
                           [0..0];
29                     break;
30         };
31     }
32 }
```

Code 6: CodAL code example of an instruction execution in the pipeline EX and WB stages.

The final part of the workflow, after testing the CA model to confirm functionality, is to let Codasip Studio interpret the code and produce synthezisable VHDL or Verilog. This code is autogenerated and is usually rather difficult to read, but for reference Code 7 is the Verilog that is generated from the final register value assignment in the previous example. Worth noting is that the registers are here seen as D-flipflops and signals are routed according to if they are inputs or outputs from the register.

```
1  assign r_pdcp_entity_D = ((ACT == 1'b1) && tmp_var) ?
       {{30{1'b0}}, {r_pdcp_entity_Q[169:138],
       tmp_conv_BITWISE_AND[17:0],r_pdcp_entity_Q[121:0]}} :
        {202{1'b0}};
```

Code 7: Example of autogenerated Verilog code.

Chapter 4
Processor Architecture

This chapter will cover important parts of the processors architecture and motivations behind its design choices.

4.1 Pipeline

The processor consists of four pipeline stages each of which is described briefly below.

1. Instruction Fetch (FE)
    - Initiate a request to read instruction data at the address given by the program counter (PC) from the memory unit.
    - Determine value for the next PC, usually next address in line if there is no context switch, hardware loop or branch interference.
2. Instruction Decode (ID)
    - Read the instruction data from the instruction cache, if memory unit is not ready the processor stalls.
    - Read the register data if the instruction has any register arguments and store it in special source registers used by the execute stage.
3. Execute (EX)
    - For base ISA ALU operations described in Appendix A.1, insert source registers as input to ALU unit and store output in special result register.
    - For ASIs, execute the functional unit associated with the instruction. ASRs may be both read and written to.
    - If it is a external communication ASI, perform a read request or write data to the SoC interface ports.
    - At a load or store instruction to the data memory, initiate the memory operation.

4. Write Back (WB)

    - For base ISA ALU operation, write back what is in the result register to specified register given in the instruction argument.
    - For load instruction write back the requested data to specified register given in the instruction argument.
    - At a read request to the SoC, write data provided from the SoC to a dedicated ASR for the specific ASI.

4.2 Memory Access

The processor is using a von Neumann architecture where program instructions and data share the same memory and pathways and memory address width is 32-bits. The memory access protocol used is a AMBA 3 AHB-Lite Protocol where the processor is the master and the memory unit is the slave.

4.3 Application Specific Registers

Usually on GPPs all program data is stored in the data memory of the processor. The data memory usually consists of an external memory block consisting of dynamic random-access memory (DRAM) cells called main memory. DRAM cells are slow to read and write data from compared to other memory types, but are cheap in regards to chip area due to the DRAM cell only consisting of a capacitor and a transistor. Static random-access memory (SRAM) cells on the other hand are faster to read and write from but cost more area than DRAM because they consist of six transistors. Most modern GPP and computer systems utilize both types of memory with a load/store architecture where data is loaded from the main memory into the fast SRAM registers when the data is needed. Fetching data from the main memory and loading it into registers will therefore be slow and cause delays in the processor execution. To get around this problem, processor manufacturers traditionally implement fast accessible memories called caches in between the register memory and the main memory. The caches contain data previously used (temporal locality) and a block of data close to the data previously used (spacial locality). When data is being fetched from main memory the system will look and see if the data is in the cache before looking in the main memory, consequently speeding up the memory access.

On the ASIP designed in this thesis all primary processing data is stored in ASRs and not in main memory. The data structures used in the application are known at ASIP construction, therefore the data structures can be allocated directly in hardware as registers inside the processor and will be accessible directly at all times without any delay. This optimisation will eliminate the need for data caches and data memory and the load and store architecture will primarily not be utilized on the ASIP. Allocating dedicated registers for data structures will cost more in terms of area compared to storing data structures in main memory, but will increase performance considerably.

The ASRs will be accessed directly by ASIs. Which ASRs the ASI will use to read and write from will be hard coded into the ASI and won't be configurable from software. The flexibility comes from that each ASR is mapped in a register mapping structure where the ASR is divided or combined into 32 bit chunks depending on it's size. These 32 bit chunks are then accessible by the base RISV-V ISA through a specific index number. As an example; in assembly register "r36" is mapped to an ASR which holds the start address of a hardware loop for task 0 and register "r52" is mapped to an ASR which holds all error and exception flags used by the ASIs. This will enable future software changes to the ASIP because the ASR then becomes accessible from a software level with normal instructions such as addition, subtraction , comparisons e.t.c.

4.4 Specialized Instructions

This subsection will cover the different categorise of specialised instructions designed to solely target the application.

4.4.1 External Communication

The ASIP is only tasked with performing a part of the data processing in the RLC and PDCP protocols described in section 2.1.1. The other part of the data processing is performed by other hardware blocks on the SoC. The ASIP therefore needs to communicate data in an efficient manner with the surrounding hardware blocks on the SoC which leads to a big part of the ASIP's ISA consisting of instructions which read or write to the SoC. Which ASRs or hardware block that are written and read from is hard-coded into the ASI.

A specification of the data transfer protocol to be implemented on the ASIP was provided by Huawei. The protocol requires a very wide full duplex parallel port interface which enables the ASIP to both read and write many bits of data each clock cycle. A read operation from the ASIP will take two clock cycles to finish; in cycle 1 the ASIP requests data and in cycle 2 the data is transmitted on the read port. A write operation will only take one clock cycle. The two cycle read operation was what motivated the four stage pipeline design together with the load and store functionality.

4.4.2 Data Processing

ASIs tasked with specific data processing on some ASRs have their own functional unit (FU) allocated used only by the specific instruction. This design choice accelerates data processing considerably but comes with a trade-off; there is no flexibility within the functional units. If the way data is processed within the instruction needs to be changed, it has to either be replaced with a complete software implementation with the base ISA, or the ASRs the ASI is processing data on have to be altered after the instruction is executed. The total on chip area also increases due to specialized hardware has to be allocated for each instruction. An example of how a functional unit may look can be seen in the CodAL code snippet below.

```
/*
* Inputs to the functional unit.
* Inputs can originate from either the instruction's
* argument or a specific value in an ASR.
*/
static void functional_unit_example(uint2 in_data1,
                    uint2 in_data2, uint2 in_data3,
                    uint18 in_data4 , uint18 in_data5,
                    uint18 in_data6){

    // Temporary signals only used inside the FU
    uint8 signal_4;
    uint10 singal_5, signal_6;
    bool signal_1 , signal_2, signal_3, signal_7 ,
           signal_8, signal_9;

     // Stores result temporary, used later in the FU
     signal_1 = in_data1 != 1;

     // Stores result permanently in a global register
     reg_1 = signal_1;

     signal_2 = in_data2 != in_data3;
     reg_2 = signal_2;

     signal_3 = in_data4 != in_data5;
     reg_3 = signal_3;

    // Selects a certain range of bits
     signal_4 = in_data4[7..0];
     signal_5 = in_data5[17..8];
     signal_6 = in_data6[17..8];

     signal_7 = in_data5 == in_data6;
     signal_8 =  (signal_4 == 0) &&
                 (signal_6 != signal_5 || signal_7);
     reg_4 = signal_8;

     signal_9 = signal_1 || signal_2 || signal_3;
     reg_5 = signal_9;

     reg_6 = signal_9 || signal_8;
    }
}
```

Code 8: FU example in CodAL.

4.4.3 Exception Handling

Sometimes faulty data is received or some other extra processing has to be performed which is not expected to happen under normal execution. To reduce branch penalties which are a result of conditional execution, a set of generalised exception handling instructions were implemented on the ASIP which jump out of the program's main loop if an exception occurs. An ASR was allocated which holds a bitmap over error flags that are set by certain data processing instructions; if any of these bits are set when the exception instruction is called it will jump to the exception handling routine. Within the exception routine appropriate measures can be taken depending on the exception that occurred. The ASIs constructed for the exception routine were constructed with more programmable flexibility than the main program. This decision was motivated by the fact that the exception routine won't be executed on a common basis; therefore the performance reduction within the exception routine due to increased flexibility won't hurt overall performance and might result in area reduction. For example, exceptions are approximated to happen every 200 iterations so if the main program loop takes 15 clock cycles to execute and the exception routine takes 5 cycles with maximum hardware optimisation and minimum flexibility, and 10 clock cycles the other way around; the increased flexibility would only take roughly 0.2% of the ASIP's total execution time.

4.4.4 Zero Overhead Loops

Each program task has a main loop; when the task has finished it will restart and run from the beginning again. At the end of the loop it will be a branch penalty of three clock cycles due to an unconditional branch instruction having to be issued and the branch is not resolved until the execution stage in the pipeline. To resolve this, hardware loops are implemented. The start and end address of the main loop for each task is saved within ASRs by an ASI which will configure the hardware loop. Each clock cycle on the instruction fetch stage the current program counter is compared to the current running task's end address, if they are equal the program counter will be set to the task's start address instead, removing the branch penalty.

Some tasks requires conditional nested loops within the main loop that are dependent on a counter value, if the counter is zero the task should exit the loop and at each iteration the counter is decremented. This was implemented similarly to the main hardware loops as described above but in addition an ASR was allocated for the counter and is checked if it is non zero in the instruction fetch stage. This optimisation saves 4 clock cycles compared to if the base ISA had been used where a decrement instruction has to be followed by a branch if not zero instruction.

4.4.5 Zero Overhead Context Switch

As mentioned previously, the firmware running on the processor is divided into a set of tasks that are more or less independent from one another. When the ASIP is requesting access to data from an external resource it takes a varying

amount of time until that data is ready. Therefore when one task is requesting some external data it has to pause execution and wait for the data to arrive. If there exist other pending or ready tasks it is unnecessary to stall the entire processor to wait for the data to arrive. To increase efficiency the processor should therefore simply switch to a ready task if the current running task is requesting external data. On a normal GPP, context switching is usually implemented on a software level and is expensive in terms of clock cycles due to the overhead of the context switch. Normally a context switch requires the processor to save the current state of the process such as the stack variables and registers containing the stack pointer, frame/base pointer, return address e.t.c in the data memory of the processor. To work around this, the concept of each task having its own stack in the data memory is scrapped on this ASIP. Each task will have its own set of ASRs where data is stored and processed, instead of in the data memory. Primarily the data memory is not used and the program stack is shared among the different tasks. Due to these optimisations the task switching can be done completely by hardware and will be performed with an ASI in only two clock cycles.

The hardware implementation of the context switch scheduler works as follows:

Step 1: Instruction execution where the argument is the input to wait for.

Step 2: The instruction data in the fetch and decode stage of the pipeline is saved in a ASR respectively. More specifically the address of the current PC and the instruction data of the next instruction to be executed. This is saved away to later on be able restore the pipeline when the issuing task is scheduled again. The input argument which indicates what I/O event the issuing task is waiting for is also saved away in a ASR called IO_status_reg.

Step 3: Each tasks IO_status_reg is then checked to see if they are ready to execute. information about which IO request is ready is read through a dataport on the ASIP sent each clock cycle from the SoC environment.

Step 4: If there is a ready task, next task is scheduled with a static scheduling priority scheme. Task 0 has highest, task 1 has second highest and all the way down to 7 which has lowest priority. This scheduling scheme was considered sufficient for the scope of this thesis. A full simulation of the SoC enviroment has to be constructed before the real-time constraints could be fully analyzed, this was not ready within the time scope of this project therefore it was assumed that starvation would not be an issue.

Step 5: The data saved in step 2 is inserted into the pipeline stages fetch and decode the next clock cycle. The execution stage is stalled one clock cycle at a context switch. Consequently there is a two clock cycle penalty at a context switch and one if no context switch is needed.

Step 6: If no ready task is found the processor sleeps and wakes up at an IO event.

Chapter 5

Results and Discussion

This chapter will present the results obtained throughout the thesis, and discuss and compare them in relation to other current processor architectures. The ASIP will be compared in-depth to a similar GPP solution, and more generally to a HAC solution.

5.1 Comparison with ARM Microprocessors

To give a good idea of where the ASIP stands against a normal microprocessor it will be compared with the ARM Cortex-M family in terms of power consumption and area usage. The Cortex-M family is according to ARM is "optimized for cost and energy-efficient microcontrollers" and "provides low-latency and a highly deterministic operation, for deeply embedded systems." [14]. This family of processors was considered to resemble the ASIP and it's intended usage the most. The specific cores that are compared to the ASIP are the highest and mid-tier processors; the M7[15] and the M33[16]. The processor characteristics in the tables below are retrieved from ARM's offical webpage.

Table 5.1: Cortex-M7 [15]

ARM-Cortex-M7

Transistor Technology (nm)	40	28	16
Gate Voltage (V_g)	0.99	0.81	0.72
Dynamic Power (μW/MHz)	58.5	31.8	18.5
Floor Planned Area (mm^2)	0.105	0.052	0.028

Table 5.2: Cortex-M33 [16]

ARM-Cortex-M33

Transistor Technology (nm)	40	28	16
Gate Voltage (V_g)	0.99	0.81	0.72
Dynamic Power (μW/MHz)	12	3.8	3.9
Floor Planned Area (mm^2)	0.028	0.014	0.008

5.1.1 Power Consumption and Area Usage

Table 5.3: Power consumption and area of the ASIP at different stages, all synthesized at 400 MHz and 7 nm.

	Non-Optimized	Optimized	Optimized Without Task 3	Estimated Total
Combinational Area (μm^2)	1738	1771	1573	2571
Non-Combinational Area	1186	1189	1097	1989
Buffer/Inverter Area	244	253	216	573
Total Cell Area (mm^2)	0.00292	0.00296	0.00267	0.00412
Power Consumption (mW)	13.958	13.963	13.871	14.331

The optimized values seen in Table 5.3 are with the optimizations described in section 3.2.5. The estimated total is an extrapolation of the results of the optimized synthesis with and without task 3; if we assume that the task complexity is roughly the same. The power consumption is calculated at 400 MHz and 100% cell load where switching power (dynamic power) is >90% of the total power consumption.

To get an estimate of the power consumption of the ASIP, a number of assumptions had to be made. Since the model isn't complete and integrated into the SoC structure, it is difficult to get an accurate value for the power consumption. If the assumption is that roughly 10% of the ASIP is activated each clock cycle, the estimated power consumption for the entire ASIP is about 1.4 mW according to Table 5.3. The uncertainty is very high since a proper testbench and stimuli source is missing in the calculation but in comparison to the ARM 16 nm core in Table 5.2, which is the most similar in design to the ASIP, it should consume around 1.6 mW at 400 MHz; not too far off from the ASIP's result. To get a properly measured value of the ASIP's power consumption, the design needs to be completed, with all seven tasks implemented. The firmware also needs to be converted into a stimuli file that can be input into the synthesis tool to simulate what inputs the ASIP receives during normal operation.

When it comes to the estimated total area, the result in Table 5.3 is more representative of the final design, the only thing that a completed design would add is the interconnect area between the ASIP core and the SoC, which is usually very small compared to the cell area. Since the ASIP is synthesized with a 7 nm technology the respective ARM result needs to be extrapolated from the 16 nm values. In Table 5.2 the area usage is halved when going from 40 to 28 nm, and then reduced by about 43% when going from 28 to 16 nm. Decreasing the transistor size will give diminishing returns with each step because of design overheads, but

if the 0.0005 mm^2/nm reduction between 28 and 16 nm is kept, then the 7 nm technology would result in an area of around 0.0045 mm^2; this can be compared to 0.00412 mm^2 for the ASIP. Since the M-33 is in the middle of the performance spectrum, the comparison might not be entirely fair. If the ASIP is compared to the higher-end M-7 model in Table 5.1, there is a greater disparity between the results. The M-7 at 7 nm and 400 MHz would be expected to consume roughly 3.9 mW and take up around 0.014 mm^2. A comparison to a lower-end model would have been a good addition, but the difference in transistor technology and size for lower-end models was determined too high to facilitate a meaningful comparison.

The power consumption results don't necessarily give a complete overview of the behaviour of the processor. As can be seen in Table 5.10, the amount of instructions that need to be executed on a GPP is much higher than on the ASIP; therefore the execution time of the same program will be much longer. A longer execution time means that the processor needs to draw power for a longer amount of time, resulting in an overall higher power consumption. The results in Table 5.3 can therefore be seen as a static comparison; the 100% on-time power consumption of the processor. Since the ASIP acts as part of a 5G modem, the assumption is also that it will spend a majority of the time in a sleep state since there will not always be data to send or process; further reducing the overall power consumption.

The result of the optimizations descried in section 3.2.5 can be observed in Table 5.3. An interesting result is that the optimized CodAL code actually resulted in slightly more on-chip area than the unoptimized code. The most likely explanation is that the synthesis tool managed to perform the optimizations in section 3.2.5 by itself. It can be concluded it is unnecessary to perform any major hardware optimisations on a CodAL level because it can be done by the synthesis tool. As a results of this more time can be spent on development of new features rather then optimizing old code, which can increase the speed of the development process.

5.1.2 Instruction Count

In total 35 ASIs were fully implemented on the ASIP and 55 instructions in total were defined in the IA model, the result can be observed in Appendix A.2. According the design specification the ASIP is supposed to have seven different tasks when the design is finished. In this project ASIs were fully completed for three of the seven task and one was only completed as a IA model, the last task was not finished due to time constraints. The last task contained 20 ASIs.

The Firmware, written in assembly, was executed in Codasip's ASIP simulator. The ASIP is assumed to have a clock cycles per instruction (CPI) of one. The two things that could affect the CPI on the ASIP is unexpected stalls from instruction memory fetch and the SoC environment. Stalls from instruction memory is redeemed as very low due to the total program size on the ASIP won't be more then a 1000 lines of assembly, therefore a small and fast memory could be implemented to store program instructions. Stall from SoC environment is hard to estimate at this point but is assumed to be very low. The amount of clock cycles spent in each task is written in the tables below. The amount of clock cycles spent on each task was calculated from to the first instruction in FE stage

to the last instruction in FE stage. To get the time for a task to fully complete which is the first instruction in FE stage and last instruction in the WB stage, one has to add 3 cycles to each entry in the tables. The tasks was executed in four rounds each to get an overview of the performance. A task can execute with or without an exception occurring, and for each data request to the SoC environment a context switch may occur dependent on if data from SoC environment is read or not. These are dynamic events and are not controlled by the ASIP. Therefore the number of executed instructions was counted for two separate cases. The first case is when a task is making a context switch at each external data request, results is seen in Table 5.5. The second case is when data is always ready and no context switch is required before the task is finished; results for this case are seen in Table 5.4. Both Table 5.5 and 5.4 list instruction count for each task separately and for all three tasks combined.

Table 5.4: Cycle count for tasks without interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	17	8
2	17	8
3	29	22
all	63	41

Table 5.5: Cycle count for tasks with interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	20	10
2	20	10
3	34	26
all	74	46

5.1.3 Equivalent ARM Instruction Count

To compare the performance increase against an ARM implementation on an instruction level a software implementation for task 1 and 3 was made in C. Task 2 is omitted because it is very similar to task 1. The difference between the two tasks is basically which registers the data is stored and written to. The software implementation was compiled using the GNU ARM cross compiler [17] targeting the Armv8-M Mainline ISA which is used by the M7 core[15]. The instructions generated for each task in the output assembly code was counted and the data path of the code was analysed to estimate the amount of instructions executed. The

context switch instruction, which is the most complex instruction on the ASIP, was written in a separate C-program and was omitted from the task 1 and task 3 program; the results can be seen in Table 5.6. For simplicity when estimating the amount of instructions executed, it was assumed that data requests to the SoC environment would be without delay and up to 320-bit could be sent in one clock cycle from the ARM processor. According to ARM's official web page[15] the M7 cortex is equipped with a 32-bit AHB peripheral port which indicates it has the capability to send 32-bits of data per clock cycle. Therefore the estimate is not completely correct but it gives a lower-end estimate of the ARM core's potential performance.

Table 5.6: Context Switch Instruction on ARM.

Instructions Generated	AVG Instructions Executed
153	292

Table 5.7: ARM Cycle count for tasks without interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	227	191
3	372	323
all	826	705

Table 5.8: ARM Cycle count for tasks with interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	1103	775
3	1832	1491
all	4038	3041

5.1.4 Instruction Count Comparison

The estimated instructions executed on the ARM machines is divided with the instructions executed for each task on the ASIP. This gives a performance increase factor in regards to executed instructions.

Table 5.9: ASIP vs ARM performance factor without interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	13.35	23.875
3	12.82	14.68
all	13.1	17.2

Table 5.10: ASIP vs ARM performance factor with interruption.

Task	Executed Instructions	
	With Exception	Without Exception
1	55.15	77.7
3	53.88	57.34
all	54.56	66.1

As seen in Table 5.10 the ASIP outperforms the ARM machines when comparing number of executed instructions by a factor of 54 with exceptions and 66 without. The main performance increase originates from the hardware context switching and without it the performance increase would only be around a factor of 13 as seen in Table 5.9. The context switch scheduler is performing a lot of data processing to select which of the 7 tasks to run next, but most of it can be done in parallel which explains why the implementation is much more efficient in hardware.

It has to be emphasized that the comparison with the ARM processors is a static comparison and it is not cycle accurate. The comparison is only in regards to the total amount of executed instructions. The instruction level comparison was done by analysing the assembly code and therefore does not include things such as:

- Stalls due to cache misses.
- Memory latency.
- Stalls due to miss predicted branches.
- Delays from external communication.

The ASIP is designed to reduce the occurrence of all the above points. Therefore a cycle accurate comparisons would most likely result in a much better performance increase with an ASIP implementation than what could be shown with a static analysis of the assembly code. Ideally the comparisons should have been made in a cycle accurate manner to include the points above; this could have been done with either the cycle accurate simulator licensed by ARM or with a physical ARM core.

5.2 Code Size

Table 5.11: Lines of code in different implementations of the ASIP.

	Lines of Code
CodAL CA Model	5323
CodAL CA Model Without Compiler	3503
Auto-generated VHDL	8919

Table 5.11 shows the amount of code that is contained within different parts of the ASIP. The CodAL CA model contains the pipeline and microarchitectural description as well as instruction semantics used by the C-compiler. In the version without the compiler, all compiler related information is removed from the code. The auto-generated VHDL doesn't contain any compiler related information so is best compared to the CA model without a compiler.

5.3 ASIP vs HAC

5.3.1 Compared to HDL Design

One of the main advantages with the ASIP design process considered is that Codasip is easier to work with than HDLs such as VHDL and Verilog. Much of the complexity introduced by a HDL is taken care of by the studio environment and relieves the developer of much responsibility. As an example, digital logic does not have to be specified as being combinational or sequential by the programmer. The studio will analyze the data paths in the source code and partition what will be sequential or combinational at RTL generation. The Studio Environment also comes with an extensive standard library and builtin functionality for bus interfaces, pipeline stalls and standard instructions which simplifies the development process considerably.

When compared with a HDL design process, the main drawback is the tools is fairly complex and it takes a bit of time to get started. For example implementing simple accelerators in a HDL could probably be simpler then implementing simple accelerators on a ASIP. If the application is small the extra overhead introduced by the ASIP design process such a compiler and pipeline directives could prove to be more complex then the application itself and therefore not suitable. With this said the ASIP design scales very well and once a structure for the pipeline

and the compiler is constructed, it is fairly easy to add more functional units and instructions into the model.

The data flow and the order the functional units is activated is handled on a software level. Therefore hardware developers don't have to think to much about connecting the different functional block together, the exception is sometime forwarding is required due to data hazards. Developers can treat each clock cycle in isolation and only has to consider the inputs and outputs for the specific instruction. The design is hence divided into independent small and simple hardware blocks which makes it easy to overlook and debug.

The CodaAl language is a modern language and is very C-like. Large parts of the CodAL code base looks similar to C-code. This make the learning curve less steep as the language constructs are recognised by both hardware and software engineers. It also minimises the gap between software and hardware implementations and enables people with a software background to a wider extent understand and read the underlying HDL code; this is very beneficial for teams combining hardware and software design.

As can be seen in Table 5.11 the amount of code required to synthesize the processor varies greatly depending on code language. Although the amount of code written is not the only factor defining development time, it can give a rough estimate on the productivity increase that an ADL such as CodAL can provide. Another benefit is that the studio provides compiler generation tools with a minimal time investment, as it is already part of the design process and not something that has to be done separately. Although the auto-generated code might not be as precise as code written by a software engineer it still can give a rough estimate on how much more needs to be done in a HDL than in CodAL to design an ASIP.

5.3.2 Flexibility

The flexibility on an HAC implementation is very low or non-existent. The ASIP on the other hand is a Turing complete computer and could theoretically perform a very wide range of applications due to it being equipped with the base ISA described in Appendix . Software changes utilising the ASRs used in the implemented tasks would probably not cost many extra instructions compared to a normal processor because no load and store has to be issued, the base instruction set can access them directly. Software implementations that are not utilising the hardware features on the ASIP will probably have to issue about the same amount of instructions as the ARM processors as seen in Table 5.9 which are about 13 times more than an ASI.

The C compiler that targets the ASIP's ISA is only recommended to be use in parts of the code that is not utilizing any hardware features on the ASIP. The main reason is the compiler has no information about the ASIs implemented and it will only schedule a ASI if it's corresponding intrinsic function is called from the C-code. Furthermore, the compiler has been restricted to not use the ASRs used by ASIs when generating the assembly code. Therefore to make changes and additions to the firmware as efficient as possible is should be done in assembly if they are targeting ASRs.

As mentioned in the constraints part of the this thesis, instructions that might

be used in future for flexibility would not be implemented. If this constraint is lifted there is potential room for flexibility improvements on the ASIP. For example there could exist a general SoC communication instruction where a command to the SoC and the registers used to store the return data could be given as an instruction argument. This would introduce flexibility if the ASIP needs to be able to send more commands or receive more data from the SoC environment in the future that was not specified at the beginning of development.

5.3.3 Power Consumption and Area Usage

To give a more in depth analysis of how the ASIP stand against a HAC implementation, a comparisons in regards to performance should ideally have been made as it was done with a GPP in this thesis. Creating a RTL for a pure HAC implementation would not be suitable for the time frame of this project. In addition the main goal of this thesis was to construct an functioning ASIP and not an HAC. Therefore a comparisons in terms of performance were omitted.

Chapter 6
Conclusions and Final Thoughts

6.1 Conclusion

The ASIP designed in the project is estimated to have around the same area and power consumption as ARM's smallest and most energy efficient series of processors. It outperforms the ARM processors with a factor of around 50-60 in terms of data processing capabilities and therefore presumably also total energy consumption. Changes in the application will result in a performance decrease on the ASIP, but it is probably going to perform better then a GPP if the changes utilize the hardware utilities on the ASIP. If the changes diverge too far from the ASIP's originally defined use case it will most likely perform worse than a GPP due to a lack of hardware optimizations such as branch prediction and data caches. It is not evaluated in this thesis if an ASIP implementation would be more or less efficient then an HAC implementation but the ASIP is definitely a much more flexible design. The ASIP design process with Codasip Studio has some extra initial complexities such as the pipeline and compiler overhead but is easier to work with once defined compared to a pure HDL implementation due to the developer being relieved of much of the responsibility. This project has also shown that an ASIP design with Codasip Studio could be very suitable for specialized applications that require high throughput and flexibility within a predefined use-case.

6.2 Codasip Studio Review

To design a processor is one of the more complex tasks an engineer can receive, and Codasip Studio has simplified it in an efficient manner. The fact that two students without any previous ASIP design experience can learn the basics of it in two to three weeks says a great deal about the tool's learning curve. The division between IA model for compiler construction and functional verification and CA model for microarchitectural definition makes the process easy to divide between software and hardware engineers. It also makes it easy for rapid design explorations.

The main problem experienced during the project was the CA model's dependency on a working compiler. The tool is designed to have a compiler ready at an early stage and it didn't really fit the project design process where an assembler would be sufficient. Another problem was that the tool was not designed to have

ASI instructions where the data processed is invisible to the compiler. For example, an instruction with no input arguments would be interpreted as doing nothing. The compiler generator would not process it and no intrinsic function would be generated. This problem was worked around with the help of the Codasip support team by "tricking" the compiler generator and feeding it false information. The conclusion of these issues is that the ASIP designed in this project was a bit outside of the general design pattern of Codasip Studio. On the other hand with assistance of the very helpful support team at Codasip all the roadblocks that accrued throughout the design process could always be solved one way or the other, adding to the flexibility of the program. Therefore the final conclusion is that Codasip Studio is definitely capable of being used as a design tool for the ASIP designed in this project and can be continued to be used for further development of the ASIP.

6.3 Final Thoughts

During the course of this five month thesis project we have learnt a great deal about the ASIP design process. It has not been without it's issues however, especially in the beginning when we where exposed to a completely new development process which none of us had any previous experience with. With hard work, determination and an intense learning curve the ASIP was successfully finalized at the end. We both studied electrical engineering at university but we have both specialized in different fields during our master's degrees. Patric has specialized in hardware and VLSI design and Lukas has specialised in embedded software. This inter-field competence proved to be essential for the ASIP design process, which required knowledge within both the hardware and software domains.

The experience we had at the Huawei office was very rewarding in terms of understanding how different it is working in an industrial research and development environment compared to at a university. As described in section 3.4 we got insight into how the product development process works; from idea inception to execution. We also got valuable insight into how it is to work in an architecture team with people with a mixed variety of competences where everyone is working together to reach the same end-goal.

Appendix A

A.1 RISC Instructions

System Calls

Instruction	Example	Description
nop	nop	Does nothing, equivalent to a stall
halt	halt	End program execution

Register data transfer

movsi	r1 = movsi 100	Load immediate value into low 16 bits
movhi	r1 = movsi 100	Load immediate value into high 16 bits
movz	r1 = movz r2,r3	Copy r3 into r1 if r2 is zero
movnz	r1 = movnz r2,r3	Copy r3 into r1 if r2 is not zero

Arithmetical and Logical operations

addi	r1 = addi r2,100	Add immediate value to r2, store in r1
add	r1 = add r2,r3	Add r2 and r3, store in r1
sub	r1 = sub r2,r3	Subtract r3 from r2, store in r1
and	r1 = and r2,r3	Logical AND r2 and r3, store in r1
or	r1 = or r2,r3	Logical OR r2 and r3, store in r1
xor	r1 = xor r2,r3	Logical XOR r2 and r3, store in r1
sll	r1 = sll r2,r3	Logical left shift r2, r3 bits
srl	r1 = srl r2,r3	Logical right shift r2, r3 bits
sra	r1 = sra r2,r3	Arithmetical right shift r2, r3 bits

Instruction	Example	Description
Comparison		
eq	r1 = eq r2,r3	If r2 equals r3, r1 is 1, else 0
neq	r1 = neq r2,r3	If r2 not equals r3, r1 is 1, else 0
slt	r1 = slt r2,r3	(signed) If r2 is less then r3, r1 is 1, else 0
ult	r1 = ult r2,r3	(unsigned) If r2 is less then r3, r1 is 1, else 0
sle	r1 = sle r2,r3	(signed) If r2 is less then or equal r3, r1 is 1, else 0
ule	r1 = ule r2,r3	(unsigned) If r2 is less then or equal r3, r1 is 1, else 0
Memory operations		
ld	r1 = ld r2	Load 32-bit value from address in r2, to r1
ldhu	r1 = ldhu r2	Load unsigned 16-bit value from address in r2, to r1
ldhs	r1 = ldhs r2	Load signed 16-bit value from address in r2, to r1
ldbu	r1 = ldbu r2	Load signed 8-bit value from address in r2, to r1
ldbs	r1 = ldbu r2	Load unsigned 8-bit value from address in r2, to r1
st	r1 = st r2	Store 32-bit value in r2 to address r1
sth	r1 = sth r2	Store 16-bit value in r2 to address r1
stb	r1 = stb r2	Store8-bit value in r2 to address r1
Branch		
jump	jump 100	Set PC to immediate value
call	call 100	Same as jump, also saves current PC in gpr_4
jumpz	jumpz r1,100	If r1 is zero, jump to immediate value
jumpnz	jumpnz r1,100	If r1 is not zero, jump to immediate value

Table A.1: List of RISC instructions divided into which functions they perform.

A.2 Application Specific Instructions

Task	Instruction	Test Coverage	Resource Read	Resource Write	Memory Access	Combinational
Task 1	Instruction 1	100%	✓			
	Instruction 2	100%		✓		
	Instruction 3	100%				✓
	Instruction 4	100%		✓		
	Instruction 5	100%	✓	✓	✓	✓
	Instruction 6	100%	✓			
Task 2	Instruction 7	100%	✓			
	Instruction 8	100%		✓		
	Instruction 9	100%				✓
	Instruction 10	100%		✓		
	Instruction 11	100%	✓	✓	✓	✓
	Instruction 12	100%	✓			
Task 3	Instruction 13	100%	✓			
	Instruction 14	100%		✓		
	Instruction 15	100%	✓			✓
	Instruction 16	100%		✓		
	Instruction 17	100%	✓			
	Instruction 18	100%		✓	✓	
	Instruction 19	100%	✓			✓
	Instruction 20	100%		✓		
	Instruction 21	100%	✓			✓
	Instruction 22	100%		✓		✓
	Instruction 23	100%				✓
	Instruction 24	100%		✓		
	Instruction 25	100%				✓
	Instruction 26	100%	✓	✓		✓
Task 4	Instruction 27	0%	✓			
	Instruction 28	0%				✓
	Instruction 29	0%	✓			✓
	Instruction 30	0%				✓
	Instruction 31	0%				✓
	Instruction 32	0%				✓
	Instruction 33	0%		✓	✓	
	Instruction 34	0%				✓
	Instruction 35	0%		✓	✓	
	Instruction 36	0%		✓		✓
	Instruction 37	0%	✓			
	Instruction 38	0%	✓			✓
	Instruction 39	0%		✓		✓
	Instruction 39	0%		✓		
	Instruction 40	0%				✓
	Instruction 41	0%				✓
	Instruction 42	0%		✓	✓	
	Instruction 43	0%		✓	✓	
	Instruction 44	0%	✓			
	Instruction 45	0%				✓
	Instruction 46	0%				✓
	Instruction 47	0%		✓		
	Instruction 48	0%		✓	✓	
Hardware Loops	hwloop	100%	✓			✓
Exception Handling	ExceptionHandler	100%	✓			✓
	JumpIfNoError	100%	✓	✓		✓
Task Switching	Instruction 52	100%	✓			✓
	Instruction 53	100%	✓		✓	✓
	Instruction 54	100%	✓	✓		✓
	Instruction 55	100%	✓	✓		
			44%	45%	16%	55%

Table A.2: List of instructions divided into tasks, and their respective resource usage.

A.3 Abbreviations

3GPP	3rd Generation Partnership Project
ADL	Architecture Description Language
AHB	Advanced High Performance Bus
ALU	Arithmetic Logic Unit
AM	Acknowledged Mode
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine (Company)
ARQ	Automatic Repeat Request
ASI	Application Specific Instruction
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
ASR	Application Specific Register
CA	Cycle Accurate
CPI	Cycles Per Instruction
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
EX	Execute (Pipeline Stage)
FE	Fetch (Pipeline Stage)
FU	Functional Unit
gNB	g-Node-B
GPP	General Purpose Processor
HDL	Hardware Description Language
IA	Instruction Accurate
ID	Instruction Decode (Pipeline Stage)
IO	Input-Output
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
HAC	Hardware Accelerator
IoT	Internet of Things
LTE	Long-Term Evolution
MAC	Medium Access Control
NR	New-Radio
PC	Program Counter
PDCP	Packet Data Convergence Protocol
PDU	Packet Data Unit
PHY	Physical Layer
RC	Reference Counter
RISC	Reduced Instruction Set Computer
RISC-V	Reduced Instruction Set Computer (5th Generation)
RLC	Radio-Link Control
ROHC	Robust Header Compression
RTL	Register-Transfer Level
QoS	Quality of Service
SDAP	Service Data Adaption Protocol

SDU	Service Data Unit
SN	Sequence Number
SPEC	Standard Performance Evaluation Corporation (Company)
SRAM	Static Random Access Memory
SoC	System-on-Chip
UE	User-End/User-Equipment
UM	Unacknowledged Mode
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
WB	Writeback (Pipeline Stage)

References

[1] Niklas A. Johansson Y-P Eric Wang Erik Eriksson and Martin Hessler. "Radio Access for Ultra-Reliable and Low-Latency 5G Communications". In: *2015 IEEE International Conference on Communication Workshop (ICCW)*. Vol. 1. IEEE. 2015, pp. 1184–1189.

[2] Sami Yangui Raissi Fatma and Frederic Camps. "Autonomous Cars, 5G Mobile Networks and Smart Cities: Beyond the Hype". In: *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (2019). DOI: https://doi.org/10.1109/wetice.2019.00046. (Visited on 04/12/2020).

[3] Wollschlaeger Martin Thilo Sauter and Juergen Jasperneite. "The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0". In: *IEEE Industrial Electronics Magazine* (2017). DOI: https://doi.org/10.1109/mie.2017.2649104. (Visited on 03/27/2020).

[4] Navrati Saxena Abhishek Roy Bharat J. R. Sahu and Hanseok Kim. "Efficient IoT Gateway over 5G Wireless: A New Design with Prototype and Implementation Results". In: *IEEE Communications Magazine* (2017). DOI: https://doi.org/10.1109/mcom.2017.1600437cm. (Visited on 04/06/2020).

[5] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Communications of the ACM* (2019). URL: doi.org/10.1145/3282307 (visited on 03/01/2020).

[6] Canalys. *Canalys Smarthphone Analysis (sell-in shipments) January 2020*. URL: https://www.canalys.com/newsroom/canalys-global-smartphone-market-q4-2019 (visited on 03/03/2020).

[7] Codasip. *Extending RISC-V ISA with custom instruction set extension*. URL: https://codasip.com/2019/05/23/extending-risc-v-isa-with-custom-instruction-set-extension/ (visited on 02/07/2020).

[8] 3GPP. *GPP's 38 specification series (2020, January 6), 3rd Generation Partnership Project (3GPP)*. URL: https://www.3gpp.org/dynareport/38-series.htm (visited on 04/17/2020).

[9] Erik Dahlman Stefan Parkvall Johan Sköld. *5G NR: The Next Generation Wireless Access Technology*. Academic Press, 2018. ISBN: 9780128143230.

[10] Michael Gschwind. "Instruction Set Selection for ASP Design". In: *IBM Thomas J. Watson Research Center* (1999). DOI: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=777382. (Visited on 04/17/2020).

[11] Masaharu Imai Yoshinori Takeuchi Keishi Sakanushi Nagisa Ishiura. "Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)". In: *IPSJ Transactions on System LSI Design Methodology* 3 (2010), pp. 161–178. DOI: https://ist.ksc.kwansei.ac.jp/~ishiura/publications/J2010-08a.pdf. (Visited on 05/14/2020).

[12] K. Keutzer S. Malik A.R.Newton. "From ASIC to ASIP: The Next Design Discontinuity". In: *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors* (2002). DOI: shorturl.at/lqHQ1. (Visited on 03/12/2020).

[13] D.R. Graham. "Incremental Development: Review of Nonmonolithic Life-Cycle Development Models". In: *ScienceDirect* (1989). DOI: https://www.sciencedirect.com/science/article/abs/pii/0950584989900499?via\%3Dihub. (Visited on 04/10/2020).

[14] ARM. *Arm Cortex-M Series Processors*. 2020. URL: https://developer.arm.com/ip-products/processors/cortex-m/ (visited on 05/19/2020).

[15] ARM Official Website. *Cortex-M33*. 2020. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7 (visited on 05/19/2020).

[16] ARM Official Website. *Cortex-M7*. 2020. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33 (visited on 05/19/2020).

[17] GNU Tools for ARM Embedded Processors Version: 4.8. *gcc-arm-none-eabi*. 2020. URL: https://github.com/intel/CODK-A-X86/tree/master/external/gcc-arm/share/doc/gcc-arm-none-eabi (visited on 05/20/2020).

Printed by Tryckeriet i E-huset, Lund 2020

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-784
http://www.eit.lth.se