Embedded hardware/software co-design methodologies for radar signal processing on multiprocessor system-on-chip

Master's thesis in Embedded Electronic System Design

KAJSA LENFORS

ALBIN NYKVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Embedded hardware/software co-design methodologies for radar signal processing on multiprocessor system-on-chip

KAJSA LENFORS
ALBIN NYKVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Embedded hardware/software co-design methodologies for radar signal processing on multiprocessor system-on-chip
KAJSA LENFORS
ALBIN NYKVIST

Supervisor: Lena Peterson, Department of Computer Science and Engineering
Advisor: Christian Takvam, SAAB AB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Embedded hardware/software co-design methodologies for radar signal processing on multiprocessor system-on-chip

KAJSA LENFORS
ALBIN NYKVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis investigates different embedded design methodologies for hardware/software co-design on multiprocessor system-on-chip (MPSoC) for radar signal processing. Two methods were introduced and investigated to perform the co-design between the processing system (PS) and programmable logic (PL) in the MPSoC. The first method was investigated to establish an efficient register-transfer level (RTL) generation tool, which was intended to be part of a complete co-design tool-chain. The second method was investigated with one particular tool, SDSoC from Xilinx, which is developed to support all aspects of co-design in one single solution.

In this project we concluded that Vivado HLS is suitable for RTL generation and could be used as part of a tool-chain for co-design. We estimated that by using Vivado HLS the total development time to realize functions as RTL decreased by approximatly 50 % compared to when implemented using HDL. Additionally we concluded that SDSoC is an efficient tool to implement all parts of co-design, including data transactions between the PS and PL. A digital signal processing system (DSP) intended for radar signal processing was implemented and tested on an MPSoC using SDSoC. By utilizing PS/PL co-design a speedup of 23.4 was achieved for the DSP system compared to when only utilizing the PS.

Keywords: Embedded, Hardware/software co-design, Multiprocessor system-on-chip (MPSoC), Radar signal processing, Xilinx SDSoC, Vivado HLS, HLS tools.

Acknowledgements

We would like to thank our supervisor at SAAB, Christian Takvam, for providing valuable help and insight during this project. His feedback and support were a huge help during planning and execution, and it contributed greatly to the quality of the project. We would also like to thank our manager at SAAB, Daniel Wallström, for making this project possible and his help with acquiring necessary tools and equipment that made every step of this project run smoothly. We would also like to thank our team at SAAB for their helpful tips and suggestions during meetings and discussions.

Finally, we would also like to thank our supervisor at Chalmers, Lena Peterson, and our examiner Per Larsson-Edefors, for all the feedback and assistance during this thesis project.

Kajsa Lenfors and Albin Nykvist, Gothenburg, June 2019

Contents

List of Figures

List of Tables

Abbreviations

AMP	Asymmetric Multiprocessing
APU	Application Processing Unit
CCI	Cache Coherent Interconnect
CFAR	Constant False Alarm Rate
DDR	Double Data Rate
DSP	Digital Signal Processing
FF	Flip-Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPD	Full-Power Domain
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLL	High-Level Language
HLS	High-Level Synthesis
IP	Intellectual property
LFM	Linear Frequency Modulated
LPD	Low-Power Domain
LUT	Look-up Table
MPSoC	Multiprocessor System-on-Chip
MTI	Moving Target Indication
OCM	On-Chip Memory
PL	Programmable Logic
PRI	Pulse Repetition Interval
PRF	Pulse Repetition Frequency
PS	Processing System
RAM	Random Access Memory
RPU	Real-time Processing Unit
RTL	Register-Transfer Level
SMMU	System Memory Management Unit
SNR	Signal-To-Noise Ratio
TCM	Tightly Coupled Memory
WNS	Worst Negative Slack

1
Introduction

The conventional radar has existed for many years and has been evolving ever since the first use during the second world war. Modern radar systems are highly digitized and utilize many digital channels resulting in a high flow of data that needs to be processed and thus the demand on throughput and latency has increased significantly. To meet these demands a high amount of computations are necessary to process the data. SAAB AB, which is a large developer and manufacturer of radar systems is planning to use complex multiprocessor system-on-chips (MPSoC) instead of Field-Programmable Gate Arrays (FPGA) and processor clusters for dealing with these challenges of signal processing.

MPSoC architectures allow advanced integration of software and hardware processing. Today's complex MPSoCs combine the software options of the on-chip processor cores with hardware acceleration in programmable logic of an FPGA [1]. These types of MPSoC architectures offer a new degree of freedom where a designer needs to determine if a function should be executed in software on the processors or accelerated in hardware [2], [3]. If a function is designated to be implemented in the FPGA parts of the MPSoC it has to be realized as register-transfer level (RTL), described by hardware description language (HDL) or by converting synthesizable software. Since developing HDL can be very time consuming, the industry moves toward using tools that support conversion of software programmed using high-level languages (HLL) into RTL. However, to be able to realize a design as RTL it needs to be described with caution and with knowledge of the limitations of hardware implementation. This opens up the possibility of developing an effective methodology to streamline the design and implementation process within any project at SAAB that aims to utilize MPSoCs for signal processing.

1.1 Problem description

The objective specified by SAAB was to investigate the implementation of radar algorithms for signal processing on a specific MPSoC platform. Since SAAB has decided to use MPSoCs for many of their systems they wanted to investigate how to utilize the full processing power of the MPSoC chip most effectively. Since the embedded processors are integrated in these chips it would be a waste to leave them unused. The current development methodology used by the engineers at SAAB is to partition functions manually early in the system design process. The functions

will either be implemented as software on processor clusters described by HLL or with HDL to realize functions as RTL to be implemented on the FPGA. With this methodology, the processors would be left unused if no effective way to utilize them is developed. Additionally, the HDL programming step is time consuming and might slow down the total design process. The complexity and slow development time of HDLs are the reason why the partitioning of functions have to be made early in the design process and why it can be problematic to make changes late in the process. Hence, to deal with this problem, an investigation and improvement of the design process of implementing signal processing algorithms on a MPSoC platform had to be performed. For this purpose, a MATLAB script modeling a digital signal processing (DSP) system was provided by SAAB. This script was to be used as base to recreate a DSP system on the MPSoC in order to develop an efficient design methodology.

1.2 Goal

The goal of this project was to determine an engineering efficient and reliable design methodology for MPSoC programming. The aim was to begin the study with focus on how RTL could be implemented efficiently on the MPSoC. Additionally, we were to determine how to utilize both the processing system (PS) and programmable logic (PL) to the full extent by investigating how hardware and software could be co-designed most efficiently. A method had to be established and tested using a MATLAB model of a DSP system provided by SAAB as case study. A desire from SAAB was to be able to decide as late as possible in the design flow if a function should be executed in software or accelerated in hardware. To fulfill this wish, it was considered a priority during the development of the co-design methodology to have this flexibility.

1.3 Limitations

When we investigated different methods of RTL implementation and co-design we did not consider any alternatives that were not available to acquire for SAAB. In other words, we only used software that SAAB could use in the future because the intent of the project was that SAAB should benefit from the findings from this thesis. For simplicity we did not consider any design options which required any other input HLL than Python, C, C++ or MATLAB.

Another important limitation was that only a small part of a signal processing system was used to do the investigation of co-design between hardware and software on the MPSoC. This was done since it would be too time consuming to implement a full signal processing system. Additionally, a simple system would suffice to illustrate the benefits of co-design, which was the purpose of this project.

Furthermore, when testing co-design on chip, we did not consider using any other resource beside the application processing unit (APU) in the PS. In a more complex

co-design the real-time processing unit (RPU) and graphic processing unit (GPU) could have been utilized along with the PL and APU, but to illustrate co-design we deemed it enough to use one PS core. All testing of co-design on chip was also limited to a Xilinx product since this was provided by SAAB.

1.4 Thesis outline

This thesis is divided into eight chapters. Chapter 2 gives a simplified explanation about the area of use for this project, radar signal processing, where the algorithms and functions that were used during implementation are explained more thoroughly. Chapter 3 shows and explains the MPSoC architecture of the platform used in this project. Chapter 4 explains some theory about MPSoC programming and proposes three different MPSoC methodologies for performing hardware/software co-design. Chapter 5 and 6 are two separate investigations that are tied to the two later proposed methodologies in chapter 4. Chapter 5 is the result of an investigation about which tool can be used most efficiently for MPSoC hardware programming in an MPSoC co-design tool-chain. Chapter 6 is the result of an investigation about how to perform a efficient MPSoC co-design with the software tool SDSoC from Xilinx. In chapter 7 we discuss the result of our two investigations and in chapter 8 we conclude this project and give a recommendation for SAAB.

2

Radar and signal processing concepts

In this chapter some basic theory about radar and signal processing concepts that will be brought up during this thesis are presented to provide a solid background to the topics.

2.1 Radar

Radar, or radio detection and ranging, uses radio waves to detect target echoes against a background of clutter and noise. Simply put, a radar consists of a transmitter that transmits a signal, an antenna for both transmitting and receiving, a receiver that receives the echo signal and a processing stage that evaluates the data, see Figure 2.1. This project did not include all functions and components in a radar system but covers parts of the conventional signal-processing stage.

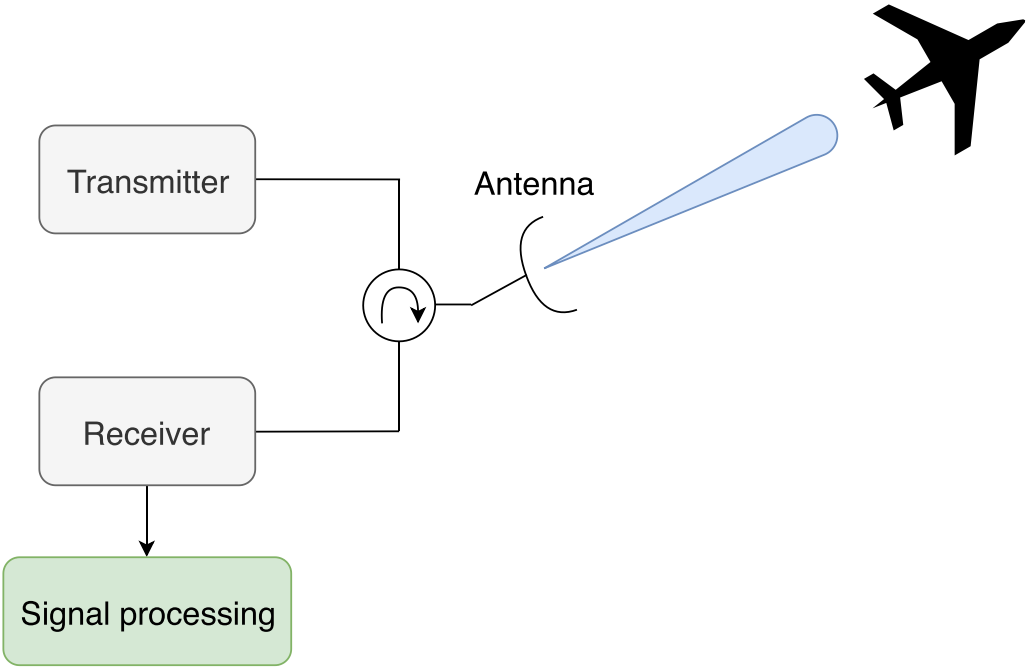

Figure 2.1: Block diagram of radar system.

2.2 Signal processing

To be able to use the radar data that is received from the antenna, noise and clutter need to be removed from the signal to improve the signal-to-noise ratio (SNR) for tracking of the targets and determining their range. The SNR improvement is performed in the signal-processing stage, which can be divided into several smaller stages as can be seen in the block diagram in Figure 2.2. Depending on the product and application the signal processing can vary and be substantially more complex. The blocks could be in a different order but in general these are the main stages that exist in a conventional radar for target tracking [4]. The signal is sampled and converted from an analog signal to digital with an A/D converter with a certain sampling rate. Pulse compression is applied to the signal to achieve a better range resolution and range performance [4]. To remove clutter which is caused by unwanted echoes from irrelevant objects, clutter filtering and Doppler processing is also applied to the the signal to be able to distinguish the targets in the target detection [4]. In the following sections the basics of pulse compression, Doppler processing and target detection will be explained in more detail.

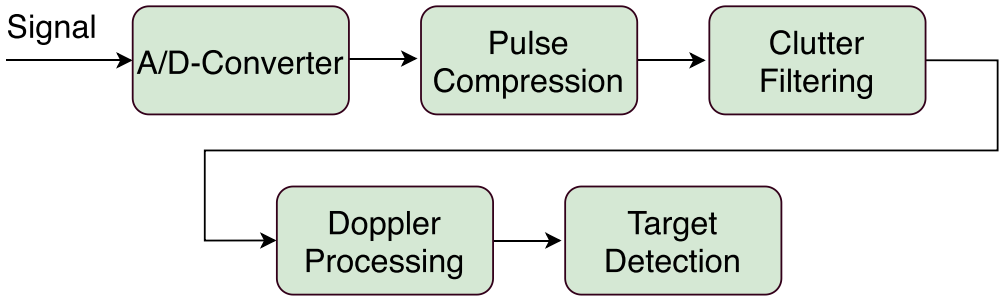

Figure 2.2: Block diagram of radar signal processing.

2.2.1 Pulse compression

Over the years the evolution of the transmitter has progressed and the requirements on the output power have become more strict. The restrictions are both because you want to decrease the overall power consumption and because lower peak power in the radar makes it harder for other actors to discover you. To decrease the average power the duty cycle of the transmitted pulse is typically increased. Long pulses, however decrease the range resolution which makes it more difficult to discriminate between targets close to each other. For this reason you typically want as short pulses as possible to maintain a high range resolution without exceeding the power limit. The pulses in Figure 2.3 illustrate the difference between the two modes. To solve this problem, a technique called pulse compression is applied to keep high range resolution while still transmitting long pulses. The principle of pulse compression is to delay the leading edge of a pulse more than the trailing edge which results in a compressed pulse.

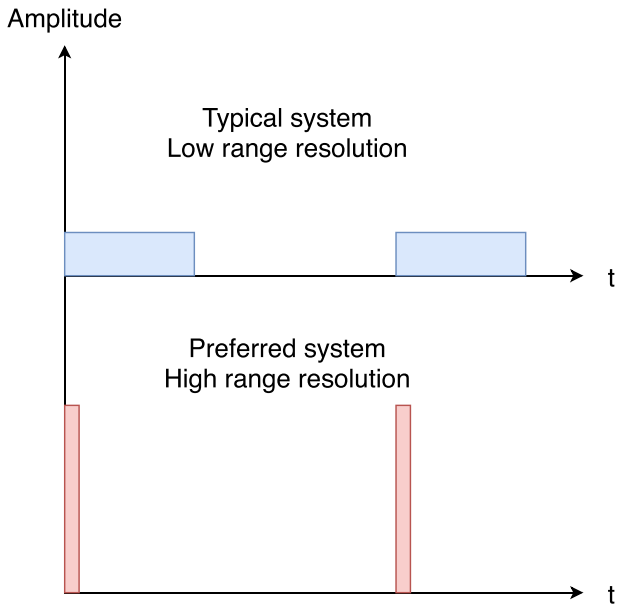

Figure 2.3: Illustration of the difference between a low and high range resolution system.

The pulse is compressed either by phase or frequency modulating the pulse and correlating it with the signal with a matched filter. To perform the discrete convolution in the matched filter either a simple finite impulse response (FIR) filter, where the modulated pulse samples are used as filter taps, can be used in the time domain or the signal and pulse can be multiplied in the frequency domain after applying a fast Fourier transform (FFT) and then reversing with an inverse FFT (IFFT). Depending on the input data, one or the other might be more desirable. FIR filters are typically used for smaller input sets due to the increasing complexity with very large input matrices and taps. For larger input sets, performing an FFT would be more efficient [4]. This thesis will only consider the FIR filter option to perform the correlation.

2.2.2 Clutter filtering and Doppler processing

As mentioned above, clutter filtering and Doppler processing are needed to be able to detect targets that may otherwise be drowned by clutter and noise. Clutter is unwanted echoes that may come from ground reflections, rain or other objects of no interest. Unfortunately these echoes usually have greater energy than those from the targets of interest which makes them hard to distinguish if the clutter is not removed first. This can be accomplished by taking advantage of the difference in Doppler frequency between the moving targets and the stationary targets. Clutter filtering and Doppler processing are techniques that use this fact.

Clutter filtering usually takes the form of moving target indication (MTI) filters which simply shift the energy of the clutter to zero-frequency and use high-pass filtering within a given range to suppress echoes from the stationary targets. The shift

can be done by knowing the speed of the antenna relative to the source of clutter and transmitting pulses that have same pulse-to-pulse phase shift [4].

Doppler processing is instead performed in the frequency domain, as opposed to MTI filtering which is done in the time domain, by using the FFT [5]. The term most commonly implies that the FFT algorithm is applied to compute the Doppler spectrum of the data. Since the difference of Doppler frequency varies for the moving targets, their energy is located in the different parts of the spectrum from the clutter energy. This information can be used to detect and separate targets. To minimize the power of sidelobes, which are due to spectral leakage, a window function is usually used together with the FFT [5]. The upside of Doppler processing is that it can provide more information about the targets from the signals, as number and velocity, which MTI does not. The downside however, is the processing complexity and increased DSP energy consumption due to a higher requirement of radar pulses [4].

2.2.3 Target detection

Target detection is needed to be able to distinguish targets from the noise and clutter and has to be efficient enough to minimize the number of false detections. The most common method for target detection is to use threshold detection to separate the targets. This method has been found to be the most optimal for performance in most cases [4]. The idea of threshold detection is to use a precomputed amplitude threshold against which the echo data is compared against. If a signal amplitude is below the threshold the signal can be disregarded as interference or clutter. If the signal is above the threshold it is marked as a detection. This method might seem simple but the problem is that the threshold is not that easy to set. As interference may vary a lot depending on the scenario, an algorithm that takes this into consideration when calculating the threshold is needed. A common method to do this is called constant false alarm rate (CFAR) detection [4].

CFAR detection takes into account the variable interference and calculates a threshold by analyzing the data itself. If a echo peak is found it uses the surrounding data as reference to determine if the peak is significant enough to determined as a detection. By using this method the false alarm rate can be maintained constant as it is continuously recalculating a threshold using interference statistics.

3

Zynq Ultrascale+ MPSoC architechture

MPSoC is a version of a system-on-chip that consists of multiple processing blocks in the processing system (PS), together with additional programmable logic (PL). In this project a Xilinx Zynq UltraScale+ MPSoC device [6], provided by SAAB, was used for testing. The Zynq UltraScale+ device (further simply referred to as Ultrascale) has a complex structure that allows for complex embedded solutions. In this chapter the architecture of the Ultrascale MPSoC is introduced. Understanding the architecture is key when programming MPSoC, since the programmer has to make design decisions such as processing location, memory utilization and interconnections between the different parts of the MPSoC device. This chapter will introduce the main processing blocks of the MPSoC, the power domains, memories and interconnection setup.

3.1 Processing blocks

The Ultrascale MPSoC features a heterogeneous processing system. This means that the architecture consists of multiple different processing components [7]. The components are divided into four major processing blocks:

- Application processing unit (APU)
- Real-time processing unit (RPU)
- Graphics processing unit (GPU)
- Programmable logic (PL)

The first three blocks, the APU, RPU and GPU, belong to the PS part of the MPSoC, also referred to as the software part in this thesis. The PL is for simplicity sometimes referred to as the hardware part of the MPSoC. An overview of the MPSoC structure, the four processing blocks together with additional important functionality such as memory, platform management and interconnect, are illustrated in Figure 3.1.

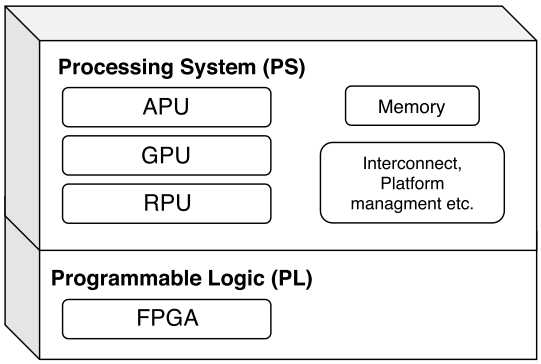

Figure 3.1: Simplified illustration of processing blocks in the Zync Ultrascale+ MPSoC together with additional system functionality.

The four processing blocks have an asymmetrical relationship, meaning that they operate separately from each other. The technique is called asymmetric multiprocessing (AMP), meaning that the different cores may run independently from each other using separate operating systems. This means that to avoid resource conflicts, mechanisms that handle data transfer and resource sharing have to be provided by the engineer when selecting design. The processing blocks mentioned above are sometimes also divided into smaller processing units within those blocks. For these internal cores the synchronization and resource sharing have to be handled, often by an operating system [8]. In this project we focused mainly on the APU when using the PS. This is due to the APU being a fast and reliable component in the PS for most general applications. In some cases other alternatives for the PS might be more suited, but to get a general idea of performance differences for hardware/software the APU will be sufficient.

3.1.1 Application processing unit

The APU is the main general-purpose processing block of the Ultrascale MPSoC. It consists of four ARM Cortex-A53 processors running at a clock frequency of 1.2 GHz. Each processor is equipped with individual level 1 (L1) cache, a memory management unit (MMU), a floating point unit (FPU) and additional connectivity systems. The Cortex-A53 processors supports both 32 and 64 bit computations. Each individual Cortex-A53 with its related components is illustrated as an instance inside the complete APU in Figure 3.2. The Cortex-A53 processors in the APU may be operated using AMP, with the cores being managed independently from each other. One of the processor cores could also be run bare metal, meaning that no operating system is used and the remaining three cores are not available for use at all. Neither of these techniques are recommended by Xilinx. Instead the processors could be run using symmetric multiprocessing (SMP), with all the cores managed by the same operating system. By using a coherency controller, or a snoop control unit, cache coherency between the shared level 2 (L2) cache is maintained among the different cores. For both SMP or AMP the cores have to be connected to a common interrupt controller to work properly. [7], [8].

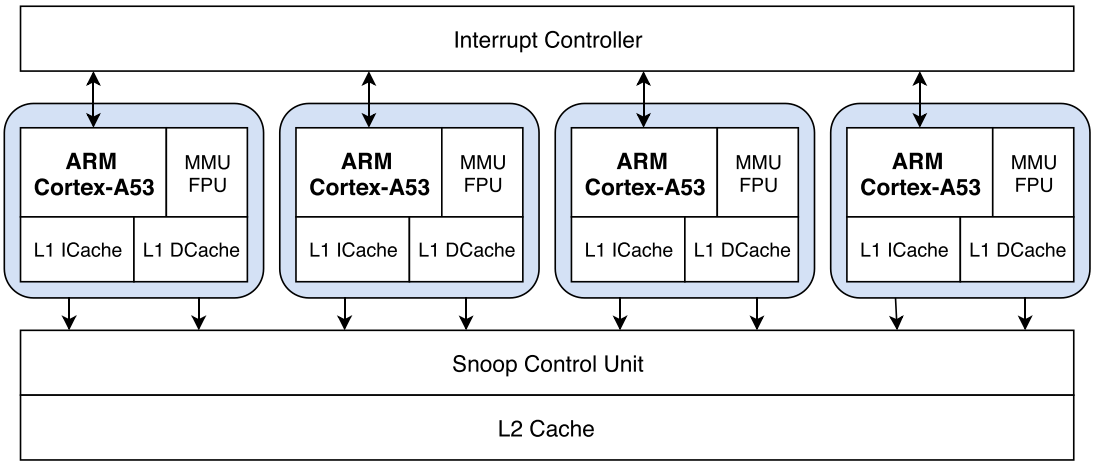

Figure 3.2: Block diagram of application processing unit in Zync Ultrascale+ MPSoC

3.1.2 Real-time processing unit

The RPU is an alternative to the APU when it comes to real-time processing and time sensitive computations. Unlike the APU, the RPU only supports 32-bit computations [7]. It has a maximum clock frequency up to 600 MHz, which is lower than for the APU. The lower maximum clock frequency does not however mean that the RPU is necessarily slower for all kinds of operations. The RPU consists of two ARM Cortex-R5 processors that are optimal for some low-latency operations. Each processor has its own dedicated cache, MMU and FPU in a design similar to the APU. Additionally, each processor has its own tightly coupled memory (TCM) with error-correcting code. The additional memories improve speed and accuracy during fast time-critical operations.

3.1.3 Graphics processing unit

The GPU is a unit specially designed for computing 2D and 3D graphics in the Ultrascale MPSoC. The MALI-400 GUP, running at up to 667 MHz, is optimal for altering memory to accelerate images intended for a display [7]. In this project the GPU was not be investigated, since the APU, PL was sufficient to run all general-purpose functions needed. The MALI-400 GPU is constructed for graphics acceleration, making the architecture not suitable for accelerating general-purpose functions [8].

3.1.4 Programmable logic

The PL, or simply FPGA, is completely integrated in the Ultrascale MPSoC. The PL is useful for processing a lot of data in parallel, especially if the programmer takes time to pipeline functionality. The complete PL part of the MPSoC consists of both logic components and communication systems used for data transfer between PS and PL [9]. In the UltraScale MPSoC platform an XCZU9EG-2FFVB1156 FPGA [10]

is used as PL. It has the logic resources presented in Table 3.1. The block random access memories (BRAM) are dual-port RAM modules that hold 36kB each. The look-up tables (LUTs), holding 64-bits, are used as small and fast memory. LUTs function as truth tables, storing the results of combinatorial logic by setting the output for every given input. They can also be used to create LUTRAM memory, a memory between LUT and BRAM in size. A flip-flop (FF) stores the state of one bit at a time and is one of the most important components in an FPGA as they are used for registers. DSP blocks are components with architecture optimized for digital signal processing, used for calculations in the FPGA [11].

Table 3.1: Resources available in the XCZU9EG-2FFVB1156 FPGA [10]

Resource	**Total**
BRAM	912
DSP	2520
LUT	274080
FF	548160

3.2 MPSoC memory

The Ultracale+ MPSoC may utilize multiple on-chip memory (OCM) components in combination with external memory. The choice of memory location is an important aspect when designing a system, since fetching data will differ in time depending on which memory and which processing unit is used [8]. The major memory components, illustrated as green blocks in Figure 3.3, are the following:

- External memory
- On-chip memory
- Tightly-coupled memory in RPU
- PL block RAM

The external memory is controlled by the double data rate (DDR) memory subsystem, and depending on the resources needed the memory size may be adjusted. In addition to the external memory there are several OCM's available on the Ultrascale MPSoC. The main PS OCM is connected to the low-power domain of the chip, with 256 kB RAM. The processing units in the PS each have their respective cache memories. Furthermore, each RPU core have additional tightly-coupled memory (TCM), which is 128 kB for each core. The PL part of the Ultrascale MPSoC has its own block RAM memory with a total size of 32.1 Mb [7], [8].

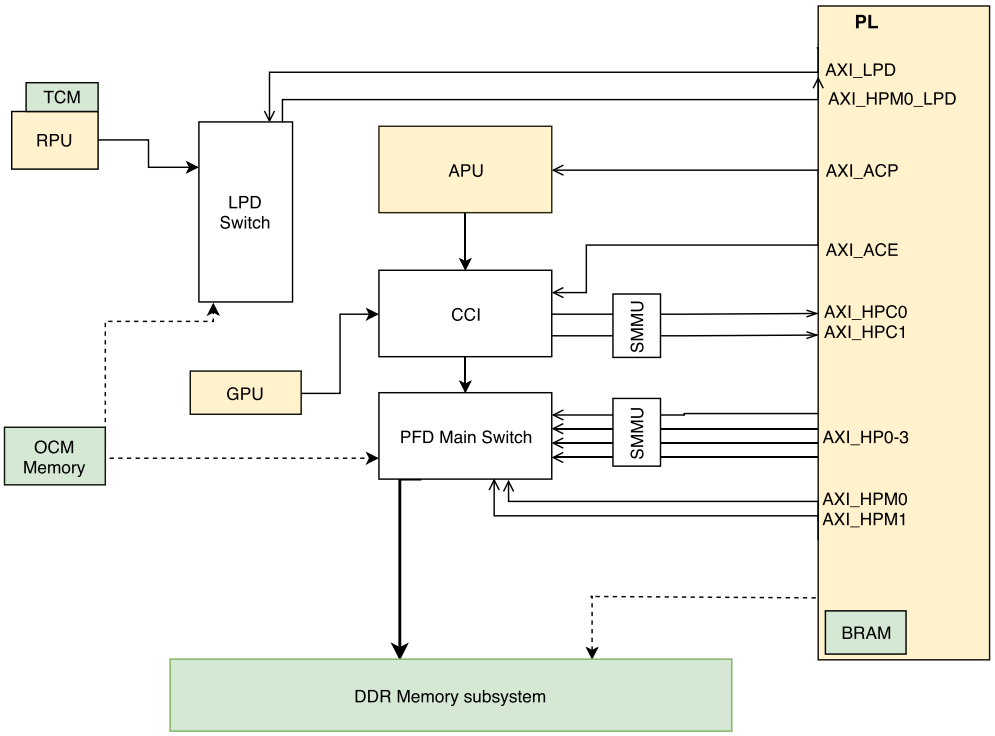

Figure 3.3: Simplified view of MPSoC device architecture. The green blocks are memories and the yellow blocks are processing units. The arrows illustrate data transportation paths between the major subsystems.

3.3 MPSoC interconnect

MPSoC is a complex device with several internal mechanisms for communication, power supply and system management. In this section we will only present the domains of an MPSoC and how they are connected. This is to give the reader an idea of the inner workings of the chip with focus on PS/PL communication.

3.3.1 Power domains

There are four independent and isolated power domains on the Ultrascale MPSoC. The domains are the low-power domain (LPD), full-power domain (FPD), PL power domain (PLPD) and battery power domain (BPD). The battery power domain is used to keep the real-time clock going, the low-power domain includes the RPU and the FPD includes the APU and GPU. The purpose of power domains is to regulate the power needs depending on what processing power is needed. The different power domains are independent from each other and the MPSoC can operate with some of them turned off [7].

3.3.2 Communication coherency

Several channels of data can be transferred simultaneously in different parts of an MPSoC device. To ensure memory coherency, a state where memory is correctly updated and synchronized, the MPSoC device uses several different methods of synchronisation. In this project we have not investigated coherency since this is managed automatically by the MPSoC. To put it simply, a large part of the hardware coherency is managed by the cache coherent interconnect (CCI), illustrated as a part of Figure 3.3. Additionally, all data transfers are done according to a coherency priority depending on what interface or port they are accessed through. The system memory management units (SMMU), illustrated as part of Figure 3.3, are used for translating virtual addresses into physical addresses and speedup the interconnect between the PS and PL [7], [8].

3.3.3 PS/PL AXI interconnect

The Ultrascale uses ARM AXI interconnect for communication between different processing units. When linking the PS and PL there are several components that work together. The data is sent through a memory system port in the host PS, transported with data movers to the PL accelerator and accessed by an PL hardware interface. The selection of appropriate AXI interfaces and data movers is an important task when designing a system on MPSoC devices, since the different combinations provide different speed, coherence and bandwidth [8], [12]. In Figure 3.3 the different AXI interfaces used for PS/PL communication are illustrated in the PL block to the right. A short explanation for every PS/PL AXI interface are described below:

Starting from the top of the picture, there are two AXI ports connected to the LPD. The AXI_LPD is directed from PL and the HPM0_LPD is directed to PL. These ports can be used even if the FPD is turned of in the MPSoC.

Next is the interconnection between the FPD and the PL. There is one AXI_ACP interface directly connecting the APU to the PL, which provides direct coherency with the L2 cache in the APU. Additionally, an AXI_ACE interconnect is likewise connected to the APU, with the difference that the ACE uses the CCI to guarantee coherency for both PL and APU. These interconnects are a good option when functions are accelerated from the APU when the data is already in the APU cache memory. If there are cache misses, it will take additional time to fetch data from OCM or DDR memory [8], [7].

There are six high performance AXI ports used to communicate with the FPD in PS. Two of these, the AXI_HPC0-1, are connected to the complete FPD through the CCI. These ports can in some cases be preferred over the ACP when communicating with the APU, since they have a higher bandwidth and they do not change the L2 cache. The four remaining high performance AXI ports, named AXI_HP0-3 ports, are used for general-purpose input to the PL. In addition to these six AXI master ports, at the bottom of the Figure 3.3 there are two inbound AXI_HPM0-1

AXI slaves that are communicating data from the FPD to PL [8].

4

Zynq Ultrascale+ MPSoC programming

In this chapter the MPSoC programming methodology is introduced. There are many important design decisions to be made during MPSoC programming, such as what processing unit to use and how to manage data transfer. In some cases the PS or PL independently might yield enough computing power for the system, while more complex systems require co-design between the PS and PL. This chapter begins with an introduction to RTL generation for the PL on the MPSoC. Next, the different programming alternatives for the PS and PL are presented together with appropriate design tools that assist during programming.

4.1 RTL realization for PL

The PL part of an MPSoC has to be realized in RTL. The RTL implementation may be done by either manually writing HDL to be realized as RTL, or by converting HLL to RTL using an HLS tool [13]. Below, the process of high level synthesis for RTL convertion is explaind. This is followed by a short presentation of the FPGA design flow and examples of appropriate tools to use for PL programming.

4.1.1 High-level synthesis

HLS starts with an abstract behavioral description, such as C or another programming language, and automatically generates an RTL description. This step can be done manually by an HDL programmer, but this can often be time consuming. In this project we estimate that an HLS tool can speed up the total development time of a hardware function with approximately 50 %. This estimate was based on a study where NVIDIA used the HLS tool Catapult from Mentor to investigate HLS efficiency [14] and used as a qualified guess of the engineering efficiency during the investigation in chapter 5. Further on in chapter 6 the impact of HLS had on engineering efficiency will be more thoroughly discussed. Different HLS tools will perform differently in terms of quality of result and speedup for total design time. However, what is important to note is that the availability of effective HLS tools makes programming faster, and even software developers might design hardware without extensive knowledge of hardware programming. HLS tools perform the five tasks of pre-processing, operation-scheduling, allocation, binding and generation, with operation-scheduling, allocation and binding being the major parts in the pro-

cess [15].

In the first step before scheduling, the HLS compiles the abstract behavioral specification code into an internal representation for the tool. This means that operations are translated into hardware-library specific formats such as shift operations instead of multipliers. The next step is the actual operation-scheduling, often simply referred to as scheduling, where the storage and functional operations are scheduled by assigning the operations to control steps (clock cycles).

In the allocation phase the types of hardware units to be used, such as registers and multiplexers, are selected from a design library. This step must fulfill the design constraints requested while still optimizing the design. An example of such a design decision could be using a certain type of adder that focuses on speed instead of minimizing resource utilization. After the allocation step all the operations are mapped to the allocated hardware units; this is called the binding step. Operations are assigned to functional units, data transfer steps are assigned to wires and buses etc. All the mapping is done with respect to the previous scheduling step. Finally the interconnections between components are set up in the data path and controller generation. The end result is complete RTL code that could be used to program an FPGA [13].

4.1.2 FPGA design flow

The PL part, or the FPGA, of an MPSoC consists of logic components, such as multiplexers and configuration memory cells, that have to be programmed with a configuration to create the desired behaviour for the FPGA. The development flow of an FPGA design consists of an RTL step, synthesis and a physical design step [13].

The RTL describes the detailed logic implementation, such as registers and flip-flops, of the entire integrated circuit. State information and what logic is needed to generate new states are described, as well as the circuit behaviour in each clock cycle. RTL is often implemented using HDLs such as VHDL or Verilog and is mapped to physical hardware using suitable tools.

Logic synthesis starts from RTL description and uses minimization mapping techniques to optimize and convert the RTL into gate-level standard-cell device-level netlists. This is called the logic level.

The **physical design** step converts the netlists to a physical representation, also called the device level. The netlist is mapped, placed and routed and a bitstream is created that is used to program the settings for all small functional elements on the FPGA.

4.1.3 MPSoC PL tools

The compilation of HLL into RTL is a practice that is useful to improve design productivity for engineers [13]. Research has been done to improve the synthesis steps and most major vendors have developed their own HLS tools for this purpose. One of the major actors is Xilinx which has developed Vivado HLS [16]. Many HLS tools use C-based languages as design entry languages. Examples of such tools are: Catapult, Synphony C Comp, ROCCC, SDAccel and Vivado HLS [13]. There are also several other HLS tools that use other design entry languages. A well known tool in this category is MATLAB HDL Coder from Mathworks [17], which uses MATLAB as input language and generates RTL. In addition to the well known vendors there are also lesser known options such as MyHDL [18], which is no HLS tool but uses Python as input language and converts it to HDL. The efficiency of these tools varies and the tools are often optimized for a certain type of platform architecture [19], [20].

4.2 MPSoC programming alternatives

There are many options for MPSoC programming. In this section MPSoC programming alternatives are explained. First, the process of selecting processing units is discussed. In some cases using only the PL might suffice, but in many cases a combination of PL and PS might be used. Next, in the end of this section, we explore three different methods for co-designing the PS and PL.

4.2.1 Selecting processing engine

To create a successful MPSoC design it is important that the right processing engine is selected for each application in the system. In some cases this might mean that the PS or PL might not be used at all. However, many embedded designs may benefit from the combined computing power of several processing units working together. The designer has to be aware of what data is being handled, how the data could be moved and stored and if the functionality of the system could be optimized by using one or more processing units. A simplified guideline for how to select processing units is illustrated in Figure 4.1.

The PL is both a fast and energy efficient alternative when selecting processing engine for an application. Otherwise, the APU will work well for most general-purpose functionality. It is possible to accelerate most of the functionality in the PL and still use the PS as a control system between the different applications in the system. If there are strict timing constraints such as in a real time system, the RPU might be a good alternative to the APU. The additional TCM for each RPU core will speed up calculations. The GPU is optimized for data related to multimedia, however the GPU is OS specific and might not always be available. For further reading about MPSoC processing unit selection, refer to [8].

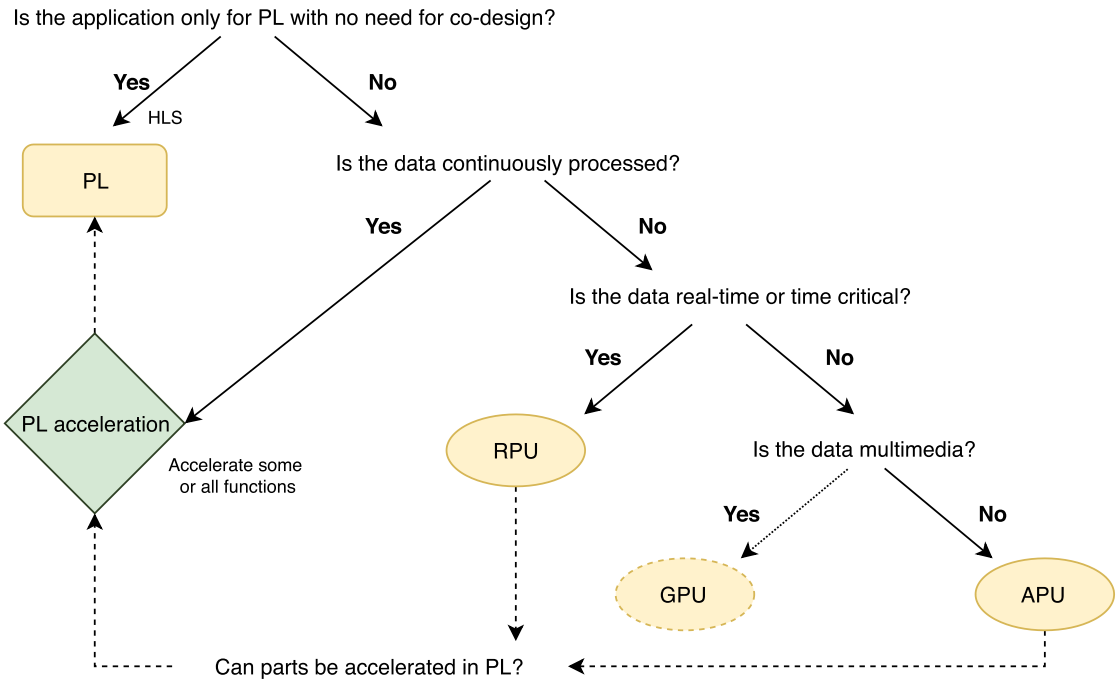

Figure 4.1: Processing unit selection for MPSoC programming [8].

4.2.2 MPSoC PS/PL co-design

There are multiple ways of programming an MPSoC. If only the PL is to be utilized, it is enough to generate RTL by following the process described in section 4.1. The RTL generation is done by using appropriate state-of-the-art tools available. If co-design of the PS and PL is required the programming complexity will increase. In some cases, the PS and PL might run separately without extensive communication between each other. In other cases, the different parts of the MPSoC will have to communicate to handle memory coherency and data transfer using AXI ports. In this project the different ways of programming were divided into three categories depending on level of automation during the design process. Either the engineer does a lot of the programming manually, or tools maybe used to streamline parts of the design process. The three different approaches are presented and explained below:

- Designing the functionality of both the PS and PL separately. HDL and HLL are coded manually for the different parts.
- Designing the complete system using an HLL and using a tool for converting selected functions into RTL to be used in the PL.
- Designing the complete system using an HLL and using a tool for RTL conversion and and data transfer management.

Using the first approach, illustrated in Figure 4.2, the engineer manually writes HDL for the PL and program the PS separately using a HLL. The engineer has to make decisions on hardware/software partitioning of functions early in the system design process. This method is a traditional way of co-designing hardware/software sys-

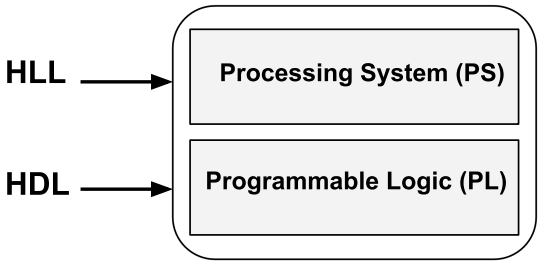

Figure 4.2: MPSoC programming design flow by manually programming PL and PL separately using a HLL and HDL.

tems. The drawback with this approach is that the engineer has to have sufficient knowledge about where a function would operate most efficiently. As mentioned before HDL can be very inefficient and time consuming during development. The option of changing the function placement later in the design process is limited since this would require complete reprogramming. This approach is most suited when a programmer has knowledge of where a function should be placed. One important aspect of MPSoC co-design is that the memory management and resource sharing of the PS and PL have to be managed by the programmer using an appropriate tool.

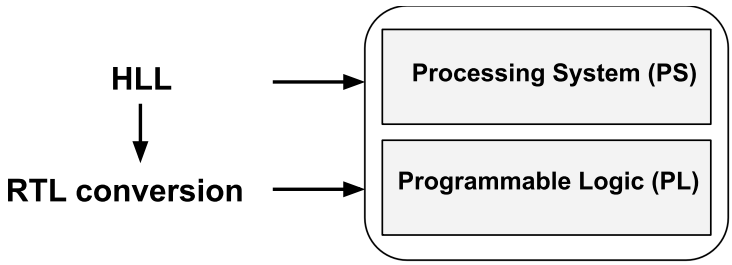

Figure 4.3: MPSoC programming design flow using a RTL conversion tool.

For the second approach, illustrated in Figure 4.3, the engineer can write the functionality of a complete system using an HLL. When the system is complete and tested the parts suitable for hardware implementation could then be converted to RTL using a suitable tool. With this approach the time consuming step of manually writing HDL will be eliminated. The partitioning of functions could consequently be made late in the design flow. In chapter 5 multiple RTL conversion tools are investigated and compared. The drawback with this programming approach is that there are several restrictions when programming with an HLL for RTL conversion. These restrictions have to be considered when designing the functionality that is to be placed in the PL. Additionally, the programmer still has to handle the memory and resource management of the MPSoC manually when using this design method. In this project we will not investigate this part of co-design further, we suggest using a design tool such as Xilinx SDK [21] for the assembly of the PS and PL functionality.

The last approach resembles the second one in the sense that the entire system

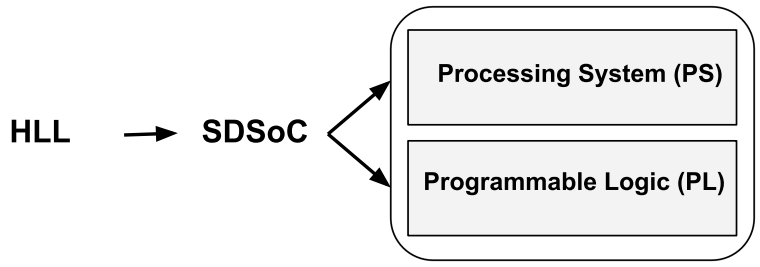

Figure 4.4: MPSoC programming design flow using a complete co-design tool.

may be designed and tested using HLL before some parts are accelerated using a state-of-the-art tool. The difference being that for this solution the memory and data management are also managed by the same tool. SDSoC is a tool specified for co-design of hardware/software systems on Xilinx chips [22]. SDSoC makes it possible to design the complete system using an HLL. The functionality that should be accelerated can be selected and changed anytime during the design process. All data transfer and memory allocation is managed in SDSoC, with the option for the designer to make changes. It is easy to test and investigate different combinations

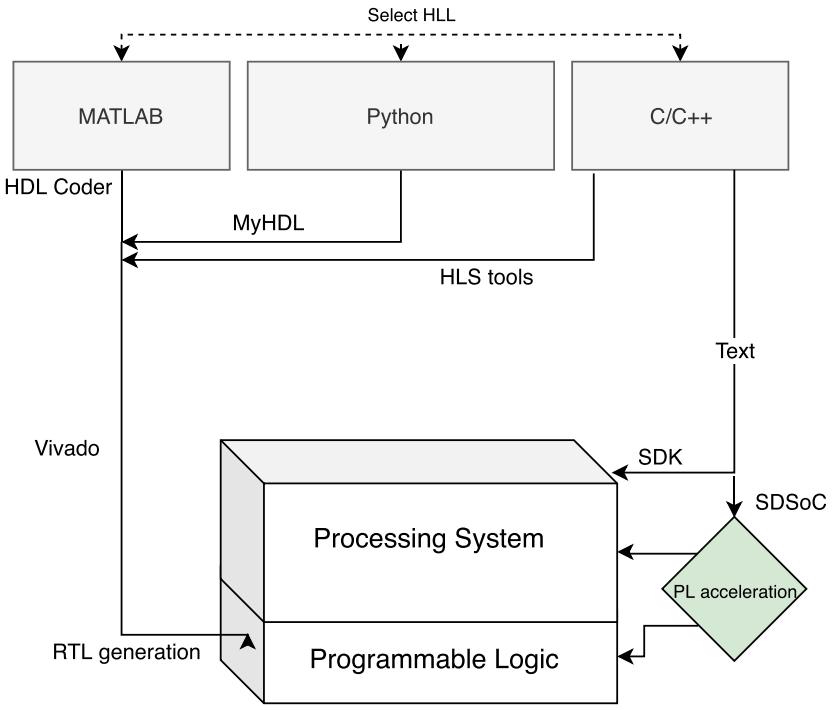

Figure 4.5: Block diagram of different tools used for MPSoC programming.

of hardware/software, which makes this tool optimal to use when a designer is not certain about what parts of the code should be accelerated. SDSoC uses Vivado HLS for the HLS parts, which makes it possible to use both programs in combination when designing and testing completely new systems. In this project SDSoC was investigated and evaluated for co-design in chapter 6. For this approach, no other

tool was investigated due to the time consuming complexity of SDSoC and the lack of equivalent tools for Xilinx products currently on the market.

To summerize the programming alternatives for MPSoC, a block diagram of several promising options investigated in this project are illustrated in Figure 4.5. In this project the first step was to investigate the RTL generation for PL programming. The optimal tool for this conversion could then in turn be used when following the second co-design approach mentioned above. This was followed by a investigation of SDSoC, the third method, as a complete co-design alternative.

5

Investigation of MPSoC hardware programming tools

In this chapter different tools used to implement RTL in the PL of the MPSoC are discussed. The purpose of this investigation was to establish the optimal tool appropriate for co-design according to the second co-design method discussed in section 4.2.2. This chapter starts with an explanation of the model used for the investigation, followed by a section about what tools we decided to evaluate more closely in this project. The resulting RTL implementations from the different tools are compared and the most suitable tool for PL programming is established. This chapter focuses only on the PL part of the MPSoC, while we investigate the co-design of the PL and PS more closely using the third co-design method in chapter 6.

5.1 Method for evaluating tools

To evaluate the different alternatives a function had to be designed using different starting point HLLs with an identical design style and functionality. An FIR filter was selected as the first component to be compared using the different tools. The FIR filter was chosen as it is a very common component in radar signal processing, for example when implementing matched filters for pulse compression. An FIR filter was considered sufficient enough for this investigation compared to designing a complete DSP system. Since the investigation was also based on research during the prestudy we found that a simple design was enough to investigate performance and flexibility during development for the selected tools. Using this FIR filter as a pulse compression model, the different tools could be evaluated depending on estimated design time, flexibility, FPGA resource usage, power consumption and timing. Timing performance was measured in worst negative slack (WNS) which is a measure of how long a result is ready before the next clock pulse arrives. Hence, higher WNS means that the clock frequency could be further increased. The design time and flexibility considered factors like language flexibility, how much time a redesign would take (spinaround time) and limitations of using the different starting point languages. The resource usage of the finished RTL implementation was established using Vivado Design Suite [23]. It is important to note that since the different solutions were based on FIR filters created in various languages there is a risk of inaccuracy when evaluating the different tools. In other words, a high-level description of the filter will not be the same for all options as there are various ways of describing an FIR filter which may lead to an unfair comparison. This had to be

considered during the evaluation. The exact filter specifications used for the FIR filter are presented in section 5.1.1.

5.1.1 Model for evaluation

To mimic an FIR filter used for pulse compression in radar signal processing systems a linear frequency modulated (LFM) chirp pulse was constructed to be used for matched filtering (correlation) as this is a commonly used waveform in radar systems [4], [5]. The LFM pulse can be described mathematically with equation 5.1 where β is the sweep bandwidth of the pulse and τ the duration [24], [25]. The rectangular envelope function, $a(t)$ is described in equation 5.2. For simplicity the pulse was sampled into $N = 21$ samples between $-10/f_s \leq t \leq 10/f_s$ to be used as filter coefficients for the matched filter. The step size is $1/f_s$ where f_s is the sampling frequency. This order was chosen due to the rising size of the RTL implementation for higher filter orders. The coefficients are presented in Table 5.1 and the pulse is plotted in Figure 5.1. Note that only the real part of the coefficients was used due to the increased complexity of FIR filters with complex filter coefficients.

$$x(t) = a(t)e^{j\pi(\beta/\tau)t^2} \tag{5.1}$$

$$a(t) = \begin{cases} 1, & 0 \leq |t| \leq \tau. \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

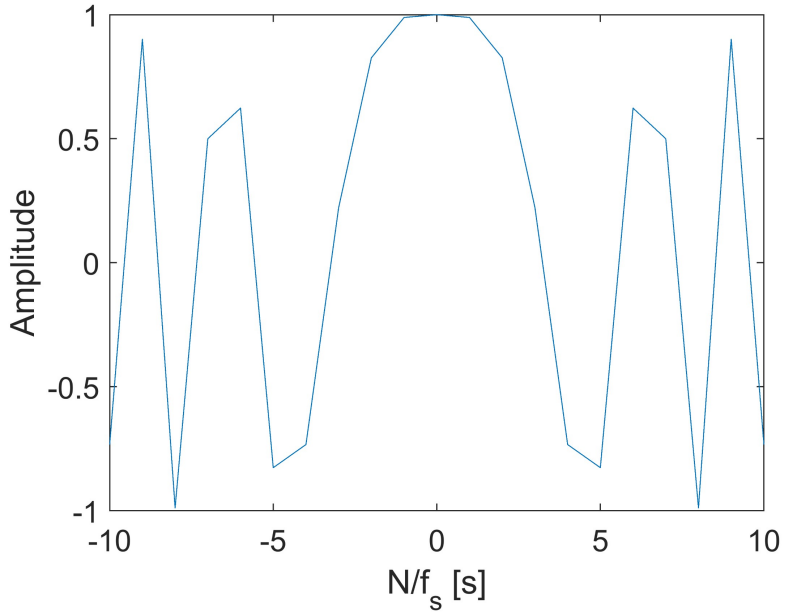

Figure 5.1: LFM chirp pulse for matched filter.

The filter coefficients and input were to be represented with 16,2 (14 bits for the fractional part, 1 for integer and 1 for sign) fixed point representation. The output

Table 5.1: Filter coefficients for FIR filter

Filter Coefficient #	Value
1	-0.7331
2	0.9010
3	-0.9888
4	0.5000
5	0.6235
6	-0.8262
7	-0.7331
8	0.2225
9	0.8262
10	0.9888
11	1.0000
12	0.9888
13	0.8262
14	0.2225
15	-0.7331
16	-0.8262
17	0.6235
18	0.5000
19	-0.9888
20	0.9010
21	-0.7331

width was set to 32,4 to avoid losing resolution from the multiplications in every stage of the filter. The filter needed to have a latency of maximum one clock cycle to make sure it was either a fully parallel or pipelined filter architecture. The timing constraint was specified to 2 ns which corresponds to a 500 MHz clock.

5.1.2 Tools selected for evaluation

As mentioned above there are several ways of converting a pulse compression model using an FIR filter into RTL. The possibility of converting a MATLAB model directly to HDL using MATLAB HDL coder was one promising option due to the advantage of having finished functions and blocks for different components. It seemed like a robust tool especially useful for quick RTL implementation for existing MATLAB functions [26]. Another promising alternative was to manually write the function in C/C++ and convert the design to RTL using Vivado HLS, which is a commonly used HLS tool that performs well for RTL conversion [27]. MyHDL was an interesting option due to the flexibility of the Python language. No further tools were investigated in practice for this step, both due to time limitations and limited availability of tools at SAAB. A summary of the selected conversion tools for evaluation is shown below:

- HDL coder - MATLAB to RTL
- MyHDL - Python to RTL
- Vivado HLS - C/C++ to RTL

5.2 Evaluation of HDL coder

HDL coder is a MATLAB addon from Mathworks [17]. Often during system design the functionality is tested in MATLAB before implementation in another language. Instead of reprogramming after testing and using an additional tool for generating HDL from a C/C++ source, HDL coder can make synthesizable code directly from a MATLAB function or Simulink model which can be implemented on an FPGA platform. The main advantage of using HDL coder is the ability to quickly generate HDL code from a function or Simulink without any additional steps to RTL implementation. By doing this you open the possibility of making a full design of a system entirely in MATLAB and being able to generate HDL for the functions partitioned for the PL and running the remaining parts could be run on the PS.

5.2.1 Restrictions of HDL coder

HLS tools will always be restricted in the sense of what can be synthesized or converted, HDL coder is no different. Since a relatively simple design was made and tested with HDL coder, not that many restrictions were encountered. However, it was noticeable that the user has to be very restrictive when designing a MATLAB function with HDL coder. Unless the I/Os are manually serialized the tool will generate an RTL structure that is not clocked but instead a parallel structure without any process statements. The user also has to be restrictive with control flow statements. The following statements are not supported for HDL conversion, **while, break, continue, return**. **For** loops always have to be bounded. If the design is instead made in Simulink, there are multiple predefined HDL blocks that can be used for the design. These blocks are easily configured and optimized for RTL implementation which for example does not require any manual serialization. So, if the functionality that is to be realized in RTL can be described with predefined HDL blocks it would be preferable to design it in Simulink. The possibility of extending the Simulink model with HDL compatible MATLAB function blocks makes it even more flexible. For that reason it would be more advantageous to use Simulink for the main design as much as possible as this is the less restrictive option of HDL coder.

5.2.2 Creating an FIR filter using HDL coder

To evaluate HDL coder, both a MATLAB function and a Simulink model of an FIR filter were used to generate HDL. Due to the requirements of a sampled system with a latency of one clock cycle with a continuous stream of input, there is a need to process and output one sample every clock cycle. For this reason, a Simulink model was more preferable to model this kind of sampled system. The model used for evaluation is illustrated in Figure 5.2 which is a predefined block of a discrete FIR

filter that can be configured to fit the specifications in section 5.1.1. The output from HDL coder resulted in an HDL solution that could utilize different clock rates. The other approach where HDL is generated directly from a MATLAB function does not give the same kind of HDL design that is of interest as the resulting RTL implementation is not a clocked module but merely a function where the sample rates is not possible to modify by the user.

Another advantage of modeling the system in Simulink is that all components that have HDL support can be configured. For example, the discrete FIR filter component that was used for evaluation can be changed to different implementation architectures, such as fully parallel, fully serial, partially serial etc. Since a latency of one clock cycle was required a fully parallel architecture was selected. Simulink blocks are also flexible in the sense of deciding I/O type (int, double, fixed-point etc.) and the resolution which determines the bit width of the I/O which was set to fit the specifications. The design can also be optimized for a certain FPGA platform and clock rate to be able to generate optimal HDL for the given platform and meet timing constraints. The Zynq Ultrascale+ device was chosen as platform and the specified clock rate of 500 MHz was set for the evaluation. The resulting HDL code was synthesized and implemented in Vivado Design Suite with the same constraints and the resulting parameters of interests is presented as **HDLcoder** in Table 5.2. Worth mentioning is that the implementation utilizes 19 DSP blocks which corresponds to one per every tap except for 0.5 where the value is shifted one bit instead of multiplied.

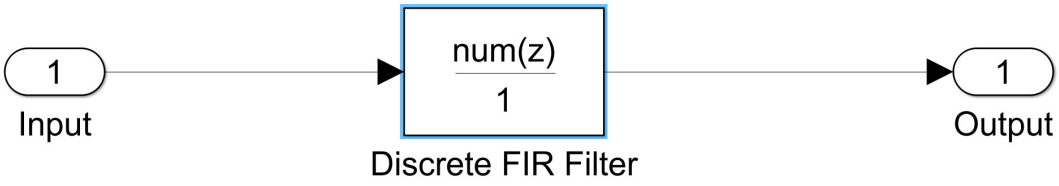

Figure 5.2: FIR filter model in Simulink

5.3 Evaluation of MyHDL

MyHDL is an open source tool used for producing synthesizable HDL from a Python based design [18]. The option to use Python was promising since it is a very high-level programming language that is especially useful for testing functionality quickly. MyHDL was investigated both due to its starting point language and due to the interest expressed about the tool by employees at SAAB. However, MyHDL is not an HLS tool and it does not take into consideration any hardware platform or timing constraints for synthesis purposes. What MyHDL does is turning synthesizable Python code into VHDL and verilog which can be used for implementation in a synthesis tool. For that reason MyHDL might be very flexible but the resulting outcome of the design might not be the most optimal. MyHDL is not intended merely for RTL implementation but could also be used for modelling and simulation

of hardware designs for functionality testing and verification before converting it to HDL [18].

5.3.1 Restrictions of MyHDL

MyHDL works by using resumable functions in Python as generators to model hardware concurrency [28]. Therefore, a top function which returns generators has to be described in order to describe a hardware module. This approach together with classes that support and implement hardware description concepts such as communication between generators and bit oriented operations makes it possible to model features that can be realized in hardware. However, to produce MyHDL code that is convertible to HDL there are restrictions and coding style rules that have to be followed:

- Generators have to be created using any of the MyHDL decorators (**instance, always, always_seq, always_comb**). Where **always_seq** and **always_comb** can be used to manually declare sequential or combinatorial logic in processes that require a sensitivity list.
- Not all object types are supported. Supported types are the regular Python types **int, long, bool, enum** and introduced in MyHDL **intbv** which is to define unsigned and signed values which needs to have a defined bit width.
- For assignment to non-variables inside processes (<= in HDL) the signal must be declared followed by *.next* (ex. *a.next = 1*).
- Process parallelism is limited as you need to declare all processes as individual generator functions. For optimal speed you might want to be able to unroll loops and perform several iterations realized as processes in parallel but MyHDL does not have an automatic way to utilize this kind of optimization. For this reason you might need to unroll and optimize code manually to meet certain constraints such as timing.

Note that there are other restrictions and rules that may not be mentioned above as these were not encountered during a not too profound investigation of MyHDL during this project. For more information and reference see [28].

5.3.2 Creating an FIR filter using MyHDL

To evaluate MyHDL the FIR filter specified in section 5.1 was modelled using the tool. The module was described as a function with the specified I/O plus a list of filter taps and value to decide the bit width of the I/O. To implement a fully parallel architecture a generator with the **always_seq** decorator was used to calculate a new output every clock cycle. The resulting VHDL code was synthesized and implemented with the timing constraint of 2 ns (500 MHz) in Vivado Design Suite and the resulting outcome in terms of the parameters of interest is presented as **MyHDL** in Table 5.2. The resulting implementation does not utilize any of the DSP blocks but rather implements all mathematical operations as pure logic (LUT) which is a lot slower which is why this design fails the timing constraint. The resulting VHDL code is however, very readable and straight forward which could be useful for creating testbenches.

5.4 Evaluation of Vivado HLS

Vivado HLS [16] is a Xilinx high-level synthesis tool which is optimized for Xilinx FPGA devices. Vivado HLS accelerates RTL design and implementation by enabling creation of RTL from C, C++ and System C specifications. It has the advantage of being highly integrated with Vivado Design suite which has been used for resource, timing and power measuring to compare different designs. Another advantage is the accelerated verification using C/C++ testbenches and automatic RTL verification from generating HDL testbenches for simulation which speeds up the design process.

5.4.1 Restrictions of Vivado HLS

There are, as with all HLS tools, some restrictions on the input design to Vivado HLS in order for the code to be synthesizable [29]. The constructs not supported by Vivado HLS are mainly related to properties such as dynamic memory allocation and dynamic variable sizes. Such constructs of the C language are not supported since it is impossible to implement on hardware. The input must have the following restrictions:

- Constructs must be of bounded size as no dynamic memory allocation may be used.
- No system calls to the operating system may be used. The entire functionality of the function must be described.
- No objects may be created at run-time, it is therefore impossible to use recursive functions.

In addition to the unsupported constructs there are several small restrictions that have to be considered when programming in Vivado HLS [29]. Components has to be designed in a certain way so that the design synthesizes correctly. The following list introduces some of the the most important design choices and good practices when programming for Vivado HLS:

- A C++ class cannot be the top-level function for synthesis. However the top-level function may consist of an instance of a class.
- The top-level function cannot be restricted, therefore no static top-level functions will synthesize.
- Pointers are supported by Vivado HLS, however arrays of pointers may not point to other pointers.
- Size all function arguments correctly, preferably by using Vivados own fixed point data types in the ap_fixed.h library.
- Make sure that Vivado can determine the exact number of times a loop will run to correctly estimate latency.
- Standard Templete Libraries in C languages often use dynamic memory allocation and cannot be synthesized in Vivado HLS. In practice this means that concepts such as vectors do not work and have to be replaced by fixed sized arrays.
- If the output of the top-level function is an array it has to be passed by reference to the function. The top-level function cannot return a pointer to an array.

5.4.2 C,C++ or System C as starting point

Vivado HLS is compatible with several C languages. With C programming the control over memory functionality is more advanced and it is consequently closer in design to the final RTL implementation. However when writing in C the programmer must have more advanced knowledge about hardware. When using C++ as a starting point many high level C++ concepts are available and supported by Vivado HLS. C++ offers a high level of abstraction for the programmer since no clocks or hardware descriptions are necessary. Another alternative is to use System C, a language based on C++ with added libraries and timing options. It uses modules which communicate by ports as building blocks, and even though it has syntax similar to C++ it still mimics HDL style programming since the timing has to be described by the programmer. Unlike C and C++ it supports the use of multiple clocks in the design [29].

In this project C++ was used as starting point since the benefits of object oriented programming made it favorable over C, and it is less complex for the software programmer than System C. Regarding multiple clocks, this is still possible to achieve by creating different components in Vivado HLS, exporting them to Vivado and manually putting them together as a complete system in the top level where several clocks may be implemented.

5.4.3 Design optimization in Vivado HLS

Vivado HLS enables many options for manual optimization and customization of the hardware implemented design. This can be done by utilizing predefined HLS pragmas which will be given as an directive to the compiler to synthesize the code or parts of it in a different way. The suggested steps to perform and use these directives are to first make the design, validate the C design and make sure it goes within the restrictions mentioned in section 5.4.1 to make sure it is synthesizable, then optimize it in the steps described in Figure 5.3 [30].

For initial optimization, interfaces can be defined for the I/O ports of the design by using **pragma hls interface** which will let the user specify how the RTL ports are created by manually assigning I/O protocols.

Pipelining for performance can be done by using **pragma hls pipeline** and/or **pragma hls dataflow**. The pipeline directive is used to allow concurrent execution within functions or loops, whereas dataflow is used at a higher level to allow loops and functions to execute concurrently.

Optimizing structures can be needed if the design is problematic to pipeline to achieve a required performance. In some cases the code might have to be redesigned but there are pragmas which can perform these optimization without any modifications. For instance, the design might have very large arrays which will slow down the performance due to limited data access with block RAM (BRAM) which has a limited number of ports. **pragma hls array_partition** can be used to solve this

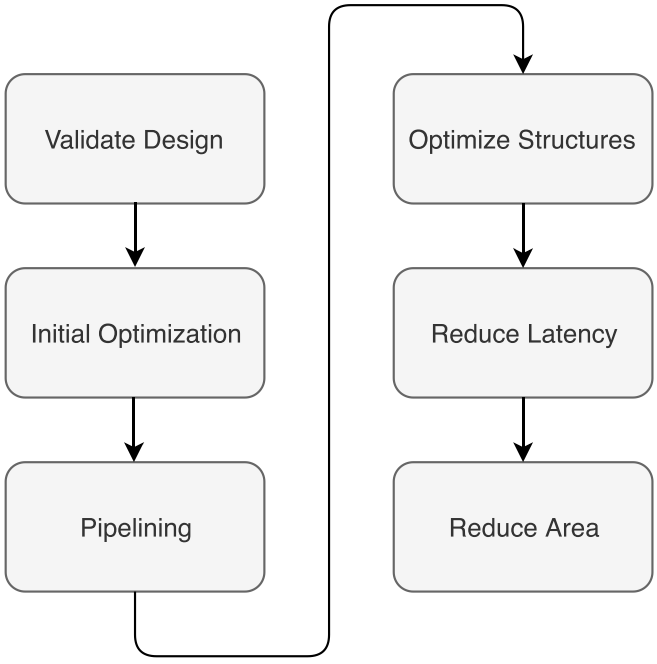

Figure 5.3: Vivado HLS Optimization Stages.

issue by partitioning arrays into smaller segments or individual registers to improve data accessing.

In many situations latency might be limiting factor to a high throughput system. The compiler in Vivado HLS always seeks to minimize the latency as much as possible. However, the directive **pragma hls latency** can be used to inform the compiler minimum and maximum constraints for the latency of a certain loop or function to improve performance.

Design area might be a limiting factor as the logic utilization could be too large for a design to implement on chip. To solve this you either have to make changes to the overall design or optimize how the logic is distributed to minimize area. Vivado HLS have several pragmas which can help to utilize resource usage such as **pragma hls resource** and **pragma hls allocation** which can improve resource sharing. The downside of these directives is that it might reduce performance.

The pragma directives described in this section are only a handful of the total amount of directives implemented with Vivado HLS. The ones described are some of the most important and basic directives used for optimization which were encountered during this thesis. Although, due to the fact that there were no resource shortage for the relatively small design that was implemented in this project, the reduce area stage was never encountered. For further information and reference for all HLS pragmas see [30].

5.4.4 Creating an FIR filter using Vivado HLS

When analyzing the design flow of Vivado HLS the FIR filter described in section 5.1.1 was designed with two different implementation styles. The filters were designed with C++ and were created as classes, where a static instance of the different class was created in each design top layer.

In the first implementation style, all the input values were shifted into and through an array while being continuously multiplied with the filter tap values. This is a common and straightforward way of designing an FIR filter. The design was optimized using the **pragma hls pipeline** command described in section 5.4.3. After synthesis this resulted in a design with 18 DSP block usage. This design is represented as **VivadoHLS_array** in Table 5.2. For this design you would expect 21 DSP blocks to be used. The three missing DSP blocks is due to DSP blocks not being used for the tap values 0.5, 1 and 0.5 which do not need multipliers as shifting logic results in the same functionality.

The second version of the filter was designed by creating a convolution matrix where all the states of the convolution could be stored between the input of a new value. This style uses a technique where the old calculations are shifted through the matrix at a new input. This design resulted in more information about the state of the filter when the design was pipelined. The synthesis result is presented as **VivadoHLS_matrix** in Table 5.2. In this design only one DSP block was created for each unique tap value besides 0.5 and 1 which results in 8 DSP blocks. This meant that Vivado HLS utilizes that the filter coefficients are symmetrical to save resources. This design of the FIR filter also resulted in the fastest implementation. The point of this filter was to illustrate that it is not only the tool but also the initial design from the programmer that affects the outcome of the HLS.

5.5 Evaluation summary

In order to compare the tools another FIR filter was implemented, using Xilinx finished intellectual property (IP) FIR compiler [31], to be used as benchmark. The functionality and tap values of the FIR filter were given to the tool and a finished FIR filter design was directly generated as an IP block. This filter, presented as **IP_Benchmark** in Table 5.2 was then used to compare the resource utilization and speed of the other filters.

Table 5.2 shows a final comparison of the FIR filter design implementation results in terms of resource utilization, timing and power consumption on the FPGA of the MPSoC platform. It was concluded from the data that the designs made in Vivado HLS are the most resource effective and fastest while keeping a significantly lower power consumption than the remaining two options. The design made with HDL coder has a low resource utilization in terms of LUT and FF's but has the highest use of DSP blocks. The usage of DSP blocks is, however, often preferred as they are more optimal for calculations in terms of speed and power. But as we can see for

Table 5.2: PL resources, timing and power comparison of FIR filter implementations using different tools. **VivadoHLS_array** and **VivadoHLS_matrix** are different implementations made in Vivado HLS.

	DSP	**LUT**	**LUTRAM**	**FF**	**WNS**	**Tot. Power**
IP_Benchmark	11	121	81	762	0.549 ns	0.694 W
HDLcoder	19	391	0	344	0.022 ns	1.013 W
MyHDL	0	2472	0	368	-2.221 ns	1.084 W
VivadoHLS_array	18	260	8	501	0.393 ns	0.688 W
VivadoHLS_matrix	8	730	311	705	0.914 ns	0.691 W

the matrix design in Vivado HLS and the FIR compiler IP benchmark, the number of DSP blocks needed could be lower due to the symmetrical filter coefficients. MyHDL doesn't utilize any of the DSP blocks as it only uses LUT and FFs for the design. This is due to that the design cannot be easily optimized or pipelined automatically by pragmas or options. As a result, the implementation is very slow and has a higher power consumption in comparison to the remaining alternatives.

To achieve a more optimal solution with MyHDL a design very close to manually optimized HDL would have to be done which contradicts the whole purpose of using the tool. What is positive with MyHDL though is that it is easy to learn, use and has a low spinaround time. It does not however have the same degree of freedom for optimization as the remaining options which make it least useful for complex designs. HDL coder is also easy to use but has higher degrees of freedom for optimization. MATLAB functions can be used together with hundreds of predefined Simulink blocks in the HDL coder library to model a hardware system. The blocks can easily be configured for optimization, pipelining and customization which makes it a better candidate for more complex designs.

Vivado HLS has the highest complexity and has a very high degree of freedom and flexibility as you can control many aspects of your design with different pragmas and coding styles. The two designs made in Vivado HLS had the most optimal resource utilization and could be clocked higher than the remaining two while having the lowest power consumption. Surprisingly, the design using a convolution matrix has both better timing and DSP usage optimization than the benchmark filter IP. Worth mentioning as well is that depending on the coding style of the designer, the solution can be significantly different in terms of resource usage and timing which can be seen for the two Vivado HLS designs. One negative aspect of Vivado HLS was that we had to put slightly more time into reading all the manuals before being able to understand and use the tool. However, we were not able to do an exact or reliable investigation of the difference in learning time between the tools since we had different background knowledge of the HLLs required and the environments when starting the investigation.

In conclusion, MyHDL is very simple and could easily be used to produce smaller less complex HDL modules with lower requirements on performance. HDL coder is

more complex and has more options for optimization and configuration but does not achieve the highest level of performance. Vivado HLS on the other hand achieves the most optimal implementation for all parameters but is harder to use due to its complexity. But due to the flexibility, options for optimization and high degree of freedom which leads to very high performance, Vivado HLS stands out as the better alternative for hardware programming. Notice that there were other promising options for HLS tools [13] that were not practically tested in this investigation. However, Vivado HLS was still one of the most promising options according to previous studies [26], [27].

From this investigation we can establish that Vivado HLS can be used as part of a co-design tool-chain according to the second method in section 4.2.2, where three different methods for MPSoC programming were presented. Since Vivado HLS only generates RTL, all communication and all connections between processing units have to be managed by the programmer. In this project we did not investigate further into co-design using this method. To do the co-design the programmer could export the design as an IP block from Vivado HLS, after which a combination of Vivado and Xilinx SDK could be used to connect the IP to hardware and create applications to run on the PS.

Additionally, we have not compared the different tools in terms of engineering efficiency, or rather the exact development and spin-around time compared to each other. This was because we did not have enough information to fairly compare the exact development speedup compared to each other. Instead, we estimated that by using any RTL generation tool the total design time could be reduced by approximately 50 %, as mentioned in section 4.1.1. Needless to say, this will vary depending on the tool. However, a more thorough investigation about the engineering efficiency gained when using Vivado HLS compared to when programming HDL manually will be discussed in section 6.5.

6

Investigation of SDSoC for MPSoC co-design

In this chapter MPSoC hardware/software implementation using SDSoC as a co-design tool is discussed. The purpose of this investigation was to establish if SDSoC can be used as an efficient co-design tool regarding both system performance and engineering efficiency. SDSoC was the third co-design method discussed in section 4.2.2 and the only tool investigated that alone can handle implementation on all MPSoC cores simultaneously. This chapter starts with an explanation of the model used for evaluation, followed by section about design optimization. Using FIR filters as an example we illustrate how parallelism is optimized in SDSoC before we continue to present a complete DSP design implemented in SDSoC. At the end of this chapter, a section about engineering efficiency and a summary of our findings are presented.

6.1 Method for evaluating SDSoC

To evaluate SDSoC the first step was to investigate how to design and optimize a relatively simple system. The FIR filter specified in section 5.1.1 was used to investigate SDSoC optimization for controlling the level of parallelism and data transfer options. A design method for optimizing several FIR filters was concluded and the results could be used to further investigate SDSoC with a more complex system. Based on this knowledge, an DSP system containing pulse compression and Doppler processing components was designed and tested. The SDSoC tool was evaluated in terms of its ability to effectively use the PS and PL systems cooperatively and the data transfer between them. Emphasis was made to determine the user-friendliness to increase engineering efficiency. To support the findings and conclusions made of the tool, data of FPGA resource usage, level of parallelism, and the total speedup achieved for the final DSP system design were measured.

6.1.1 DSP-system evaluation model

As mentioned in the introduction a MATLAB script was used to model an DSP system that we were to recreate on the MPSoC. The script includes the creation of input stimuli as a video package of received echo data that is to be processed. The input data created contains white noise, main-lobe clutter that is centered around zero frequency and a specified number of targets. The target range, amplitude and velocity relative to the receiver can be stated to test a different number of config-

urations. Specifications of the radar transmitter and receiver can also be specified which has an impact on the amount of input data created. The script uses the input data and processes it through pulse compression, Doppler processing and target detection functions and the result can be used to calculate the target range and its velocity. The goal was to use the created input stimuli from the script and process it on the MPSoC platform to have the same functionality as the script. The theory behind the functionality has been described already in chapter 2 so this section will give a more applied description of the functionality that was to be implemented.

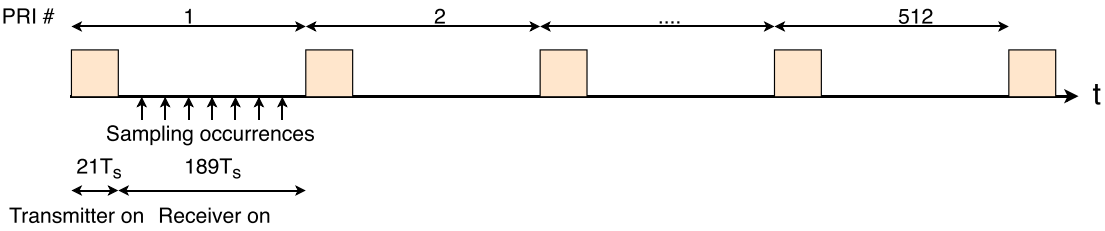

Figure 6.1: Radar sampling visualization

The input video package that is created is represented by a matrix of complex samples to describe amplitude and phase of the echo data. The number of columns is the number of pulse repetition intervals (PRI) gathered (per video package) and every column has a number of complex values which has been sampled between the pulses, see Figure 6.1. The number of samples corresponds to the number of rows in the matrix which is also called range bins. A bin length is the distance a pulse travels during the sampling interval T_s, which is corresponding to $\frac{c}{2f_s}$, where c is the speed of light. All relevant radar receiver and input data specifications that were used to create the input stimuli are stated in Table 6.1 and 6.2.

Table 6.1: Radar specifications.

Radar specifications	**Value**
Receiver sampling frequency	1 MHz
Chirp pulse bandwidth	1 MHz
Pulse repetition frequency (PRF)	4.76 kHz
Bin length	149.85 m
Pulse duty cycle	10 %
No. of samples per PRI	210
No. of PRI's	512
Video package matrix size	189×512

Figure 6.2 shows three plots of the input stimuli created. The video image shows an image created from the matrix where every value is describing a pixel. Max columns is showing the maximum value in every column (or pulse) which cannot be interpreted until the Doppler filtering has been made. Max rows shows the maximum value in every row (range bin) which gives a clear view of the two received

Table 6.2: Targets specified in input stimuli.

Target specifications	Target 1	Target 2
Range (km)	25	200
Velocity (m/s)	250	-300
Amplitude (dB)	60	50

chirp echoes from the two different targets which are similar to square pulses when displayed in logarithmic scale.

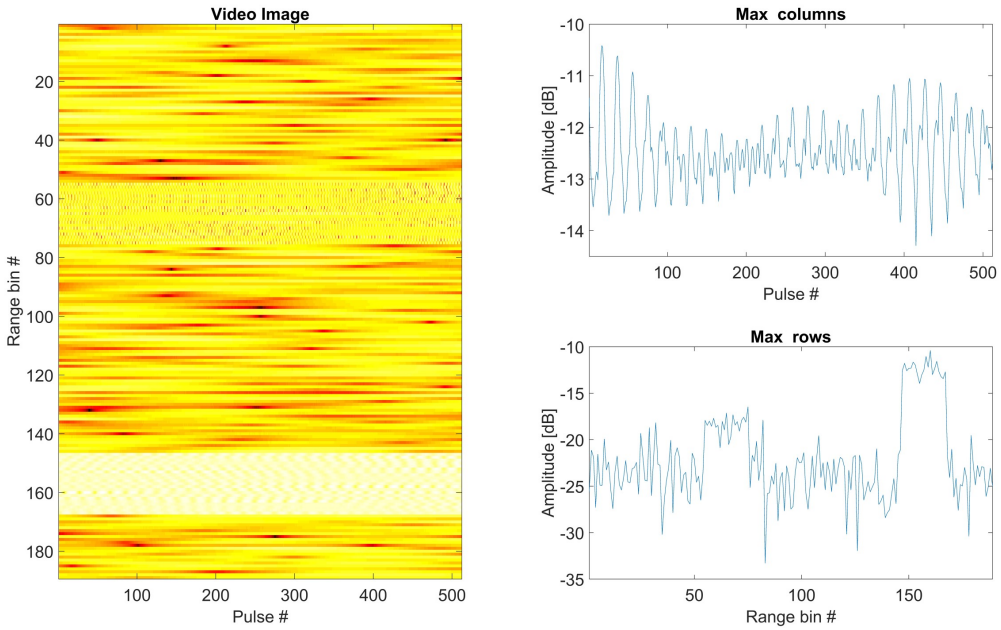

Figure 6.2: Plot of input stimuli. Showing the video image and maximum value in every column and row of the echo data.

Pulse compression is then applied to every pulse (column) using FIR filters to perform a convolution or correlation with an LFM chirp pulse in the same way as described in section 5.1.1. However, in section 5.1.1 only the real part of the pulse was used, where as in the applied case both the real and imaginary parts of the signal need to be used. The number of samples $N = 21$ was kept for simplicity to keep the not too high filter order. The new corresponding filter coefficients for the filter are presented in Table 6.3.

The Doppler processing is made by performing a spectral analysis of the data by computing the spectrum of each range bin (each row of the matrix) of the data. This computation is done by applying an independent FFT on each of the rows in the matrix. To minimize the power of sidelobes, a window function is applied before the FFT is performed. In this case a discrete Hanning window was used which can

Table 6.3: Filter coefficients for FIR filter

Filter Coefficient #	**Value**
1	- 0.7331 + j0.6802
2	0.9010 - j0.4339
3	- 0.9888 - j0.1490
4	0.5000 + j0.8660
5	0.6235 - j0.7818
6	- 0.8262 - j0.5633
7	- 0.7331 + j0.6802
8	0.2225 + j0.9749
9	0.8262 + j0.5633
10	0.9888 + j0.1490
11	1.0000 + j0.0000
12	0.9888 + j0.1490
13	0.8262 + j0.5633
14	0.2225 + j0.9749
15	- 0.7331 + j0.6802
16	- 0.8262 - j0.5633
17	0.6235 - j0.7818
18	0.5000 + j0.8660
19	- 0.9888 - j0.1490
20	0.9010 - j0.4339
21	- 0.7331 + j0.6802

be described by equation 6.1 [32] where M is equal to the size of the matrix rows ($M = 512$).

$$w[n] = \begin{cases} 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) = \sin^2\left(\frac{\pi n}{M-1}\right), & 0 \leq n \leq M-1. \\ 0, & \text{otherwise.} \end{cases} \tag{6.1}$$

The resulting video package matrix after the pulse compression and Doppler processing can then be used for tracking detection. The tracking detection in the script is used by applying an CFAR algorithm which uses the surroundings of the data to determine if the peak is significant enough to be determined a match. However, this function was not implemented in our system due to time constraints. For for that reason only pulse compression and Doppler processing were implemented. A block diagram of the operations performed on the matrix is presented in Figure 6.3.

6.2 Design optimizations for SDSoC

When a design is compiled in SDSoC the program makes an effort to select the best settings for design choices such as how data should be stored, how it should be accessed, if it should be moved and how in that case it should be transferred. Optimization of these settings is an important step of SDSoC programming because

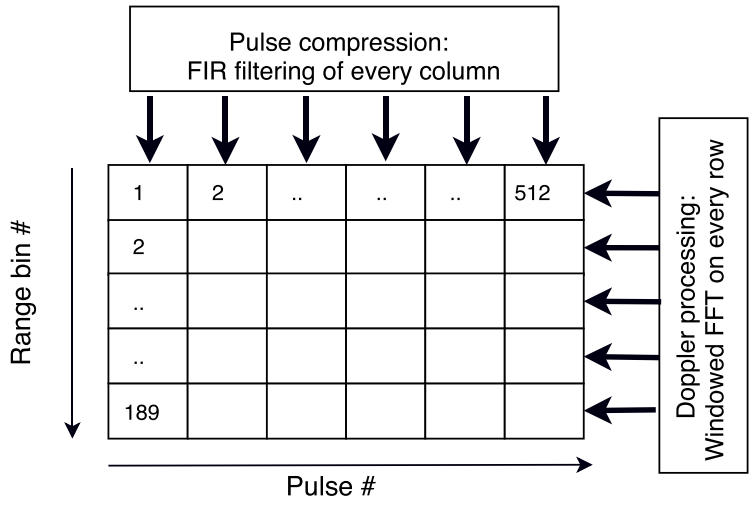

Figure 6.3: Block diagram of matrix processing

the default settings might not be the optimal solution for every design. Since SDSoC uses Vivado HLS during synthesis, the HLS pragmas described in section 5.4.3 can still be used. In addition to those pragmas there are several SDSoC specific pragmas used to override settings made by the tool to customize data management between processing units in SDSoC [33].

pragma sds async is used to synchronize a thread running in software with an accelerated function thread running in hardware. The pragma makes it possible to continue running a function in the PS that in turn has called an accelerated function in PL. The software thread continues execution until a point in the program where a **pragma sds wait** command with the correct wait-ID is placed. This is an useful pragma to synchronize hardware and software and make sure that parallelism is achieved between them when there is no data dependencies.

In many cases when data is transferred between the PS and PL the data has a sequential access pattern. This means that every value in the data set will be read and written back in the exact order they are sent, such as in an array accessed from start to end. **pragma sds data access_pattern** is used to tell SDSoC what access pattern should be used for a specified hardware function. With the default random access pattern an RAM interface will be set up to transfer data before the accelerated function start. For a sequential access pattern the data transfer will be set up with a streaming FIFO (first in first out) interface that is sent continuously when the accelerated function is running. For large arrays exceeding 16K (16384 elements) the BRAM storage is not enough to send using random access pattern. Instead sequential access pattern with the FIFO interface can be used. If random is preferred, the arrays have to be divided into smaller parts.

Data transfer and setup time is sometimes substantial compared to the actual execution time of an accelerated function. In addition to changing access pattern

between PS/PL it is also possible to change where the actual data should be accessed. **pragma sds data copy** is used to specify if the data should be copied to the accelerating PL using a data mover or if the data should be accessed directly from the shared memory in the PS. The default is to use COPY and move the data to BRAM in the PL. By using ZERO_COPY the data can be directly accessed using an AXI bus interface. It is important to note that FIFO access pattern is not possible to use together with ZERO_COPY.

There are several pragmas that can be used for additional optimization of data transfer. The **pragma sds sys_port** is used for changing system default memory port. This could be useful in cases when a high performance port is preferred to the SDSoC default coherent system port, such as the AXI_HP compared to the coherent alternative AXI_HPC from section 3.3. Additionally, data transfer can be optimized by selection the correct memory allocation for the data to be sent. There are C commands included in the SDSoC library to allocate memory. One such command is sds_alloc() which is used for telling SDSoC that data should have a contiguous allocation with cache coherent calls.

Another important SDSoC optimization is the tuning of the clock frequency used for the data transfer and the PL. SDSoC is limited in the sense that the programmer is not able to utilize the clock frequency directly by using C++ with methods such as programming their own clock dividers. However, there are a selection of clock frequencies available when functions are set to be accelerated in hardware determined by the different clock domains available on the FPGA. The data motion network clock frequency, the frequency used to transfer data, can also be changed in the settings [34]. Important to notice is that when using sequential access pattern with the FIFO interface the data motion network clock frequency has to be the same as for the accelerated logic. For further reading about SDSoC optimization refer to [12], [33].

6.3 Creating FIR filters using SDSoC

The SDSoC evaluation started with an investigation about parallelism and data transfer options using FIR filters as example functions. The same C++ FIR filter implemented using Vivado HLS in section 5.4.4 was used in this investigation. To simplify in SDSoC the FIR filter design was changed so that arrays were used as input and output instead of sequentially sending all separate values one at a time. The investigation was made in order to get an understanding of SDSoC programming with the aim of using these methods when designing an DSP system. The knowledge gained from this investigation is summarized in section 6.3.1 and 6.3.2 below.

6.3.1 Serial implementation in hardware

By running a hardware function several times in a row a serial implementation is achieved. This can be done in two different ways. One way is to call the same hardware function multiple times from the main function in the software. This method

is illustrated to the left in Figure 6.4 by using FIR filters as example functions. Another alternative is to call a single top function placed in hardware, where in turn the FIR filters are looped multiple times, illustrated to the right in Figure 6.4. Since both these methods call the same hardware function repeatedly, the same resources will be utilized in the PL for every loop. By SDSoC default, the input and output data are stored temporarily within BRAM in PL while the function executes and then it is sent back to the PS after completion.

For the first method, when random access pattern is used, the input array and output array of the filter have to complete its transfer before the next filter can be called. The data transfer will consequently take up a lot of time when this method is used. Using the second method, the data transfer could be accelerated by bursting the transfer of data to a top function. The input arrays could be embedded into a matrix and sent as a data package to the hardware. All the FIR filters are then executed in series, and the output array is then written back to the PS in the form of a data package matrix.

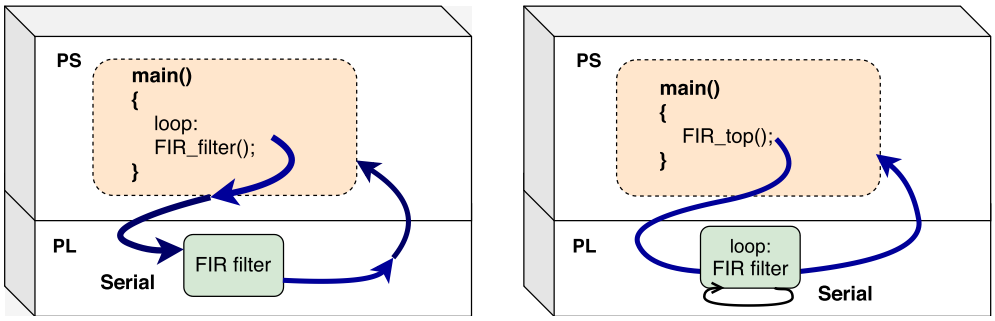

Figure 6.4: Serial implementation of FIR filters on MPSoC. To the left is a filter called multiple times in the PS. To the right is a filter looped multiple times in a top function in the PL.

An additional alternative for both methods would be to use FIFO access pattern to transfer data while the functions are running. Important to note is that there will still be a setup time in PS before data can be sent or received. The delay can make it pointless to use FIFO for accelerated functions that execute faster than the setup time. For time consuming functions with a large data set the FIFO access pattern might be preferred. Based on this, we concluded during this project that using the second method, calling a top function with FIFO, was the fastest alternative when implementing several functions in series.

6.3.2 Parallelism in hardware

Parallelism is achieved by running multiple functions at the same time. By running functions in parallel the total execution time for a system can be reduced, with the trade-off that more resources will be needed. Hardware parallelism in SDSoC can be achieved in two ways; either by calling the functions separately, or by calling a top function that in turn executes the functions in parallel. For all parallel functions in

hardware the parallelism is limited by the number of BRAM read and write ports. It is not possible to read from the same data set, such as an array or an matrix, at two places at the same time. Additionally, there cannot be more than eight BRAM-channel or FIFO-channel inputs and outputs to a hardware function [34].

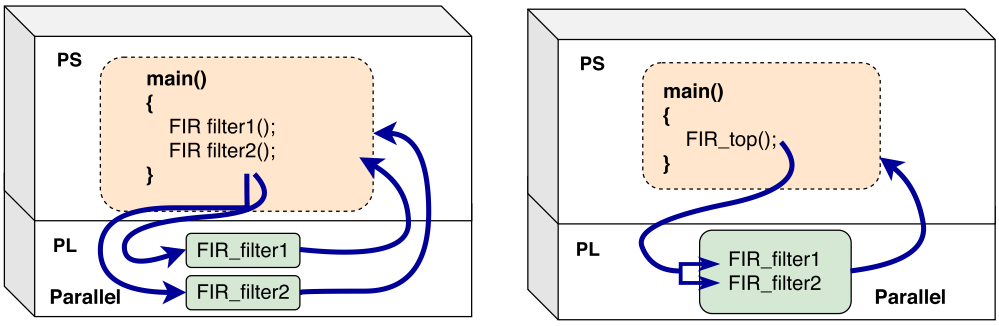

Figure 6.5: Parallel implementation of FIR filters on MPSoC. To the left two filters are called in the PS. To the right two filters are merged into one top function in the PL.

The first method is illustrated to the left in Figure 6.5 using FIR filters. One problem that we found with this approach is that the accelerated functions do not start at the same time. When using random access pattern, there is a setup delay when the PS prepares to transfer the data to and from the accelerated functions, followed by he actual data transfers before the accelerated function can start. For simple functions that execute quickly the setup delay might be substantial compared to the execution time, and the functions intended to be parallel might execute sequentially anyway. However, for time consuming functions consisting of heavy calculations the start delay is a small part of the total execution time and does not have the same impact. Additionally, time consuming functions that require a lot of data movement can be sped up by using the sequential access pattern with FIFO interface, which sends the data at the same time as the function is executing. FIFO still has a small setup delay when sending from the PS, but the actual data transfer can be done at the same time as executing the function in hardware.

For the second method, a top function can be used to burst send multiple arrays of data to the PL. SDSoC will combine all logic placed in the top function into a single solution. This method is problematic in the cases when several values have to be read from the same data input sets at the same time, which is often the case for parallel systems. This limitation also makes it problematic to send several arrays together in a matrix, since only one value in the matrix can be accessed at a time. This was not a problem during the serial implementation, where a matrix was preferred instead of using multiple arrays as input. We also found that when combining several functions in parallel in hardware the total resources used may be more then when using the functions separately. Additionally, even when using the sequential access pattern the calculations in PL did not start before all the setups for data transfer in PS was completed, making this method wait longer before anything could execute. Because of this, a top function is not the optimal solution when

implementing large-scale systems that should be run in parallel.

6.3.3 Optimizing multiple FIR filters

In this project we investigated the optimal method for executing multiple FIR filters in a time and resource efficient manner. The inputs used to the FIR filters were two matrices consisting of 512 rows of 189 complex samples, where every row in each matrix was to be executed in an FIR filter. Using the knowledge gained from sections 6.3.1 and 6.3.2 we concluded that a combination of serial and parallel execution would result in a fast system with a high throughput.

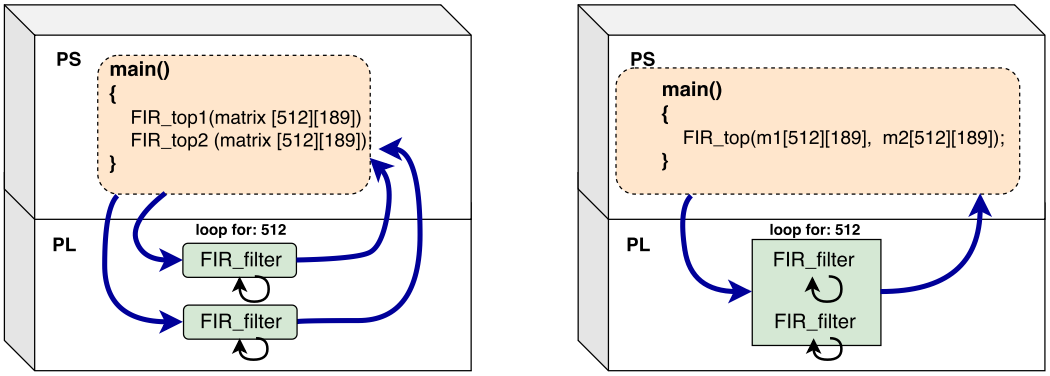

Figure 6.6: Example of multiple FIR filters executed on MPSoC. To the left two top functions are run in parallel. To the right the complete system is merged into one top function.

It was quickly established that executing 512 FIR filters in series was a solution that was both quick and resource efficient in hardware. However, since there were two sets of matrices we decided to parallelize the execution of both sets. We investigated the two different ways from section 6.3.2 to run hardware in parallel; either by using two separate top functions or combining the complete system into a single complete top function, illustrated in Figure 6.6. We used the result from SDSoC's event-tracing feature to track the two different hardware functions when the programs were executed on chip [35]. Using event-tracing we got a clear view of both software setup time, data transfer time and actual hardware run-time for each function that was accelerated to PL.

The trace of the first solution, when using two separate top functions for the real and imaginary matrices, is illustrated in Figure 6.7. The trace starts when the top function in PS calls the first accelerated function to be executed. A start command is sent to the PL and the process waits until the PL is ready to run. The setup for data transfer is then initialized and when the PS is ready the data can be transferred to the PL for the first function. Data transfer is handled with PS/PL synchronization barriers in form of "wait" commands that SDSoC automatically creates for accelerated functions. The PS waits for all the inputs and outputs to be sent and

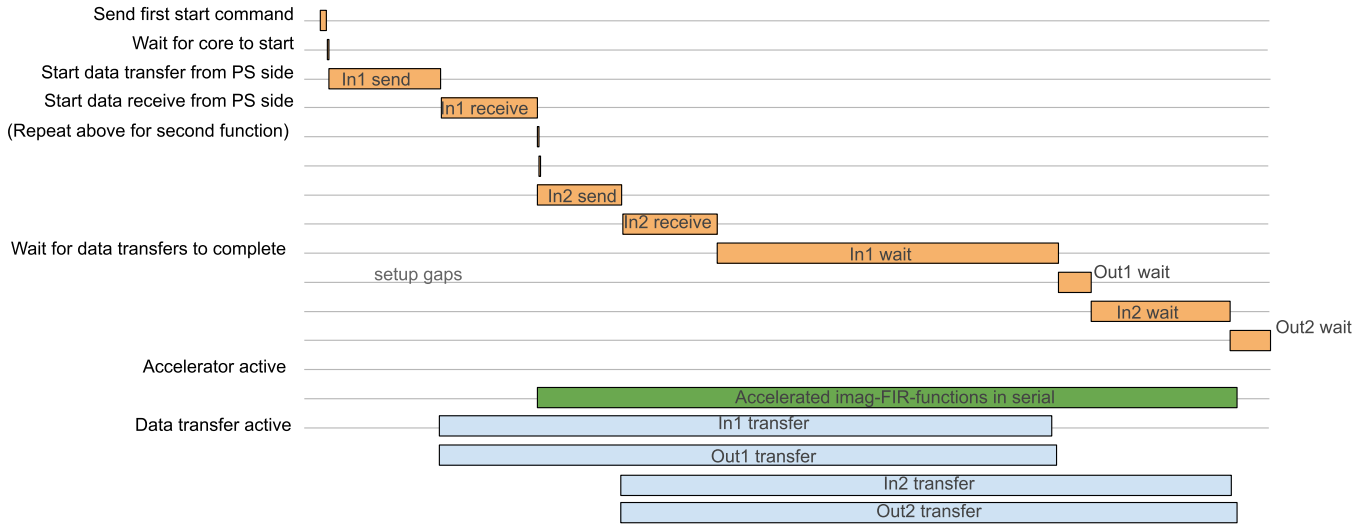

Figure 6.7: The function traces of multiple FIR filters executed on MPSoC using two top functions. The orange top boxes belong to the processing system for setting up and waiting for data transfer and synchronization. The green blocks are the active time of the accelerated functions in PL. The blue blocks at the bottom are when data transfer is active between the cores. To the left is an explanation of each step of the process. Every trace illustrated henceforth will follow this basic structure.

then the process is complete.

It is worth to notice that even if it looks like the accelerators and the data transfers are active, in reality the execution does not start until both data transfer and data receive setups are completed in the PS. The actual times where the accelerations begin are marked in Figure 6.7. This is the setup delay that is mentioned in section 6.3.2. The ”setup gap” also results in the second accelerated function starting a bit later than the first one. Since both functions have a long run-time this does not matter much, but for functions with shorter run-time this gap could remove the parallelism in the hardware.

The second solution, the trace of which is illustrated in Figure 6.8, uses a single top function to manage all FIR filters. The event-tracing graph is similar to the first solution, with some differences regarding the order of data transfers. In this solution all the data transfer and receive setupts have to be completed before anything can start executing in the hardware. As mentioned before, even if it looks like the accelerator is active the actual execution does not start before transfer to the top function is completed. All logic is then executed and the data transfer is handled using FIFO access pattern.

When comparing the two methods we found that the solution using two different top functions for parallelism was the optimal one. The results from the resource

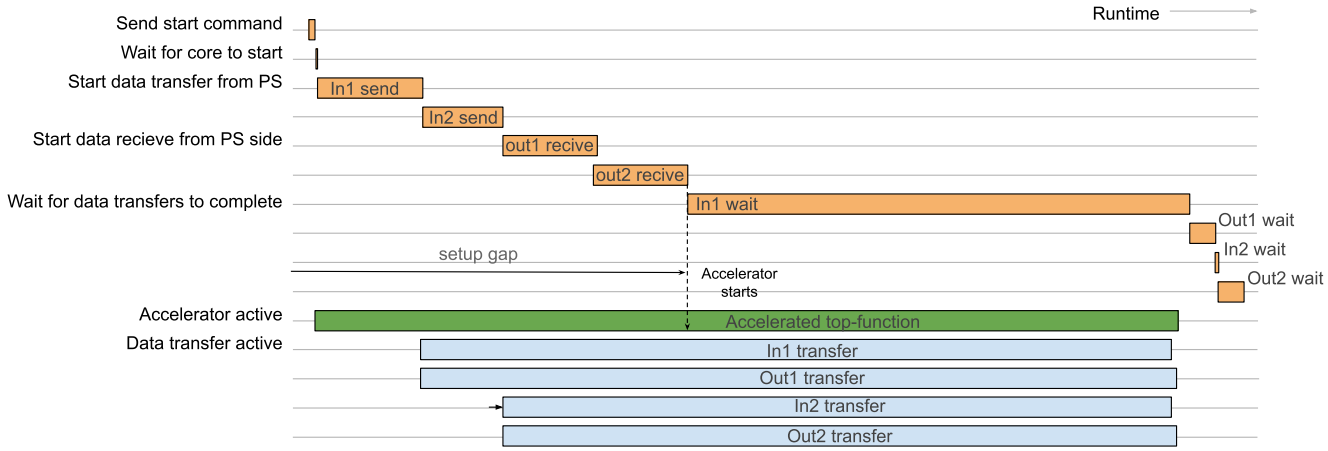

Figure 6.8: The function traces of multiple FIR filters executed on MPSoC using one combined top function.

utilization for both methods are presented in Table 6.4. As mentioned before, a combined solution need slightly more resources than a solution with two different top functions. Additionally, the total execution time for the combined top function was slightly longer than for the solution using separate top functions. This was because of the PS setup time before data could be transferred was longer for the combined solution. The final design using two parallel top functions is the design illustrated in Figure 6.6, and it is the same solution as which trace is presented in Figure 6.7.

Table 6.4: Resource utilization for multiple FIR filters using one combined or two separate top functions.

	DSP	**LUT**	**LUTRAM**	**FF**	**BRAM**
Combined top function	0.63 %	10.48 %	1.51 %	11.31 %	5.92
Separate top functions	0.63 %	7.95 %	1.85 %	6.7 %	5.92

Worth to notice is that the FIFO access pattern is a must for this design. This is because large sets of data (over 6384 elements) are transferred, making random access pattern impossible. One other option to the final design would be to send the matrices data to the PL by using a long array that contains all the smaller arrays combined. That array would then have to be divided into smaller arrays in the PL using **pragma sds array_partition** to make parallelism possible. Unlike a matrix whose memory allocation is paged, the memory allocation for an array might be set to contiguous using **pragma sds_alloc** in SDSoC. By allocating memory as contiguous, the data transfer could be a bit faster for the final design. However, this would require a lot of time consuming loops in the PL to do the array partition that outweighs the advantages.

6.4 Creating an DSP system using SDSoC

After learning how to use and optimize SDSoC for DSP purposes, a larger system was designed with the tool. The system was intended to mimic the MATLAB script which simulates an DSP system that was described in section 6.1.1. The following sections explains how the pulse compression and Doppler processing components were individually designed and then assembled into a DSP system. The system was assembled using knowledge gained from the previous sections 6.2 and 6.3.

6.4.1 Pulse compression

The bank of FIR filters that was optimized in SDSoC in section 6.3.3 could be reused for pulse compression but with slight modifications. As already mentioned, the pulse compression for this system is performed by applying a correlation of a LFM chirp pulse, whose coefficients are declared in Table 6.3, to every column of the input data. The filter coefficients were declared in two separate filter functions, one for the real and one for the imaginary coefficients. The discrete complex convolution can then be performed by using the mathematical definition of convolution and complex multiplication shown in equation 6.2 where $x[n]$ is the input, $c[n]$ the filter coefficients and $y[n]$ the output. The real and imaginary part of the output was then calculated separately in two top functions which in turn performs two separate convolutions in parallel and adds/subtracts the values.

$$x[n] = x_{real}[n] + jx_{imag}[n], \quad c[n] = c_{real}[n] + jc_{imag}[n]$$

$$\begin{aligned} y[n] = (x * c)[n] = \underbrace{((x_{real} * c_{real})[n] - (x_{imag} * c_{imag})[n])}_{\text{Real part of output}} + \\ + j \underbrace{((x_{real} * c_{imag})[n] + (x_{imag} * c_{real})[n])}_{\text{Imaginary part of output}} \end{aligned} \tag{6.2}$$

By using this approach and utilizing that the two top functions could operate in parallel on hardware, we could calculate the real and imaginary part of the output data simultaneously. The only drawback would be the small delay between the start of each function, similar to what could be seen in Figure 6.7, by using the FIFO interface for the I/Os. The individual filters still use 16,2/32,4 fixed-point representation as I/O which was specified in section 5.1.1 to avoid internal overflow. Another approach would have been to utilize the C library for complex calculations. This was however more time consuming and a higher performance due to parallelism could be achieved by the suggested approach with two top functions.

The resulting output data after pulse compressing the generated input stimuli is plotted in Figure 6.9 where we can see that the two original pulses from Figure 6.2 have been compressed and their energy is now concentrated in one range bin which results in an increased range resolution.

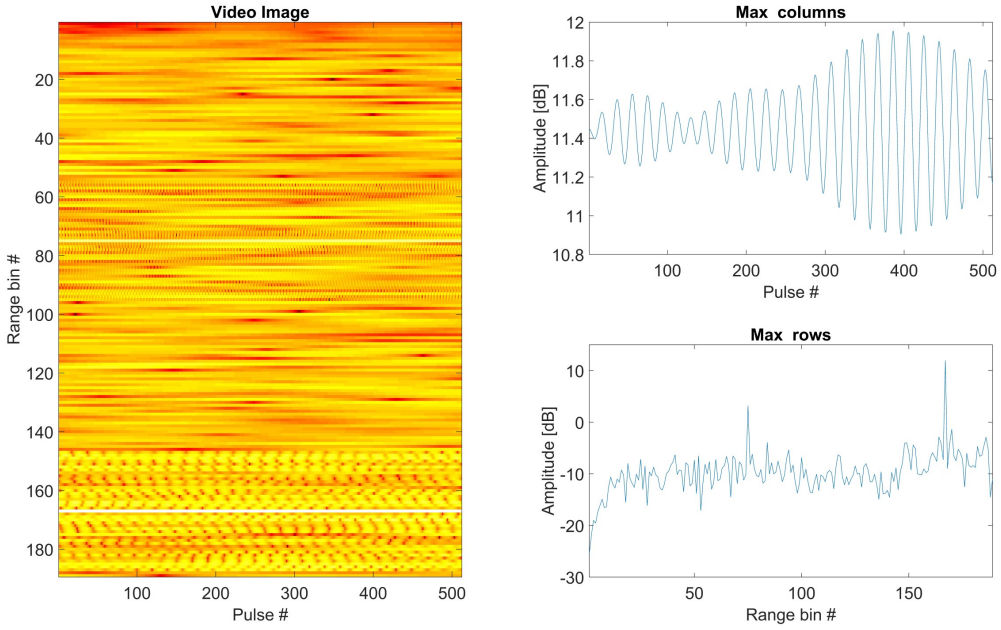

Figure 6.9: Input stimuli after pulse compression.

6.4.2 Doppler processing

To perform the Doppler processing an FFT algorithm was developed to be able to perform a spectral analysis. Since the emphasis of this project was not to maximize throughput, the FFT algorithm was not fully optimized for hardware acceleration to decrease latency but rather fully operational to be able to run it in both hardware and software. For this reason a radix-2 Cooley Tukey FFT algorithm [36] was programmed in C++ using Vivado HLS. A radix-2 algorithm needs the input length to be a power of two, which is why the number of PRI's was chosen to 512 for the input set. Since the pulse compression outputs values with 32,4 fixed-point representation, the FFT algorithm was designed with the same representation for all I/O's. But as fixed-point representation can cause overflow internally when calculating sums, the output data from the pulse compression had to be scaled to avoid calculation errors. To avoid overflow a scaling by 1024 is necessary due to the input set size of 512.

To execute the independent FFT on each row of the input matrix, a similar top function as for the pulse compression was used. The top function takes the whole input matrix and loops through the rows of data and performs a windowed FFT on each of the 189 rows. Due to the input size once again the FIFO interface had to be used as I/O protocol. Since FIFO requires the design to only access each value once, registers that temporary store the input and output values had to be implemented in the design which unfortunately increases the logic resource usage. The resulting output data can be seen in Figure 6.10 where the maximum columns now show two significant peaks that can be translated to the Doppler frequency of each target which can be used to resolve how a target is moving and calculate its

velocity.

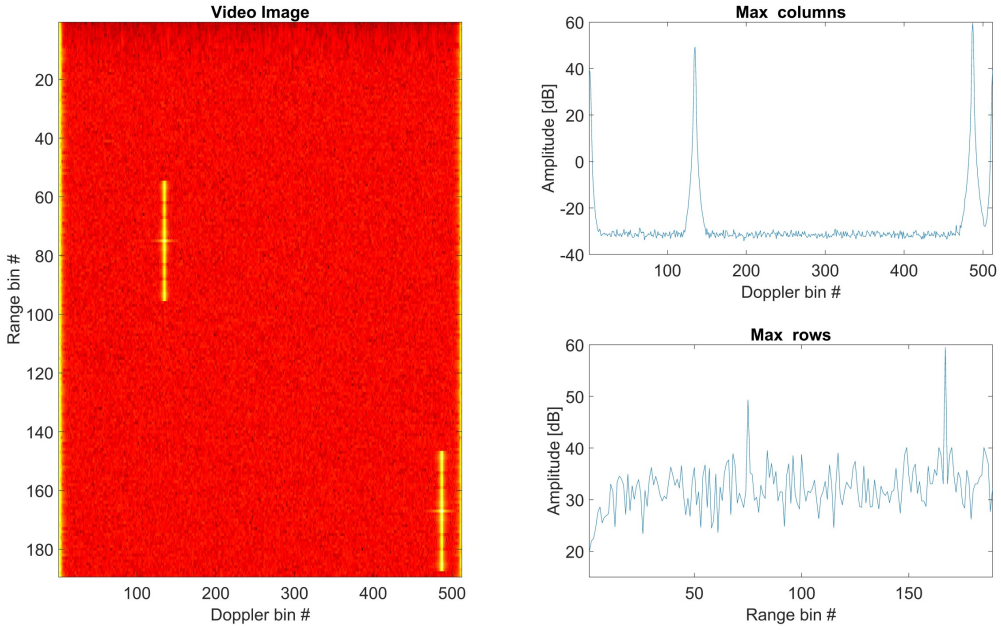

Figure 6.10: Input stimuli after pulse compression and Doppler processing.

Since the FFT algorithm is computationally heavy it is much slower than the FIR filter bank for the pulse compression. For that reason we wanted to be able to speed up the function by utilizing parallelism between the PS and PL of the MPSoC. This could be done by letting the PS perform a few of the calculations simultaneously as the remaining part is performed in the PL. This could be done by dividing the top function into two separate functions that calculate a part of the 189 rows each. By using the **pragma sds async** the PS was free to perform operations simultaneously as a function was accelerated in the PL. However, to be able to gain as much speed as possible the function which performs the processing as software in the PS had to be changed to use double-precision floating-point format instead of fixed-point format I/Os. This is due to fixed-point values being very slow to use in software in comparison to doubles. The average number of CPU cycles needed to calculate the FFT of the input matrix was measured for both the PS and PL function to decide which ratio of FFT's to run on each platform for optimal speedup. The PL function was measured to be 2.2 times faster than the algorithm for the PS, which results in an optimal data ratio of $\frac{2.2}{3.2}$ for the PL and $\frac{1}{3.2}$ for the PS which should result in a total speedup of $\frac{3.2}{2.2} = 1.45$ for the Doppler processing of the entire input set. The function was tested with the ratio which is corresponding to $\frac{2.2 \cdot 189}{3.2} \approx 130$ of the FFTs on hardware in PL and $\frac{189}{3.2} \approx 59$ as software in the PS, which resulted in a measured average speedup of 1.36 in comparison to calculating all FFTs in the PL. The reason the calculated speedup of 1.45 was not achieved was due to nonlinearites for the speedup when accelerating a function. The nonlinearities are caused by the setup gap when sending and receiving data from the PS.

6.4.3 Assembly of DSP system

For the purpose of evaluating a complete system, the pulse compression and Doppler processing components from section 6.4.1 and 6.4.2 were assembled into an DSP system using SDSoC. The ambition was to include CFAR calculations as well, but unfortunately there was no time to implement this component. The pulse compression and Doppler processing functions were prioritized in this project since they are computationally heavy, which is of interest when investigating DSP systems.

In addition to the pulse compression and Doppler processing functions, the system had to have a mechanism to flip the matrix data. Since the pulse compression is performed on every column of the data and Doppler processing on every row, the matrix has to be flipped or ”corner-turned” between the functions. This matrix flip was performed in software as a function with a regular nested loop which saves the values from every column as a row in the new matrix. Since multiple functions cannot read from the same input matrix at the same time, the flip matrix function also divides the data into two separate matrices to be able to perform the Doppler processing in both the PS and PL simultaneously. During the separation, the function also converts the values that are to be run as software from fixed-point to doubles which, as discussed in section 6.4.2, is needed to speed up the execution. The final DSP system that was implemented is illustrated in Figure 6.11.

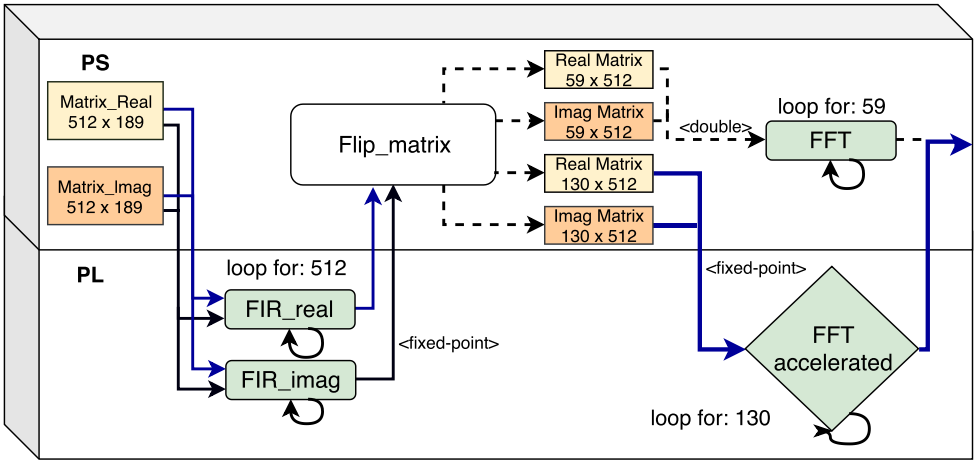

Figure 6.11: DSP system consisting of pulse compression, a flip-matrix function and FFT algorithm for Doppler processing.

Both the pulse compression and Doppler processing accelerated components were implemented on the same clock domain in the PL. Tested separately the FFT algorithm in the Doppler processing could be clocked with 300 MHz and the FIR filters in the pulse compression were possible to clock at 200 MHz. However, as mentioned in section 6.2, it is not possible to use different clock frequencies for the data motion network and components that are using the FIFO interface. This constraint unfortunately limits the components to operate on the same clock domain as the slowest component. The selected clock domain was consequently 200 MHz, which was limited by the pulse compression due to the FIR filters. The FFT algorithm in

the Doppler processing could be clocked with a higher frequency, and the limitation is unfortunate considering it has a longer total execution time.

In order to test and simulate the DSP system we used SDSoC to create several files needed to run the application on the chip. A boot file was created for booting the Linux system and programming the bitstream for the FPGA. The boot file in turn loaded an image file containing the Linux kernel to be booted on the ARM processors in the APU. Additionally, an application ELF file containing an CPU code image for PS was generated. The three files were stored on an external SD card which the MPSoC platform can read from. The input data provided in the MATLAB script was written to .txt files also stored on the SD card. The data could then be read by the APU main function into two input matrices, one for the real data set and one for the imaginary. This input setup was a temporary solution used to test the DSP functionality. In a more realistic system the input data would be streamed through an A/D converter, but this was not tested. To run the application Linux was booted from the SD card and the ELF file was executed. The output was observed on a terminal connected through an USB UART port. Additionally, using the SDSoC trace functionality the real-time functionality of the application on the PL could be monitored.

The resulting trace from the pulse compression is illustrated in Figure 6.12. It has a similar design to the one optimized for multiple FIR filters in section 6.3.3, with the difference being that a Hanning window and an extra input matrix have been added to each FIR top function. As previously mentioned, the real and imaginary data were processed separately in the pulse compression function. Since the FIR filters in the pulse compression were optimized for hardware acceleration, the complete pulse compression was performed only in the PL. This resulted in a significant speedup, presented in Table 6.6, when compared to executing it using the PS. Because of this speedup in the PL we did not consider further acceleration of the pulse compression by executing part of the system in the PL and part of the system in the PS, as done with the Doppler processing. The overall execution time for the pulse compression only accounts for a very small part (approximately 10%) of the total execution time, as can be seen in the full system trace in Figure 6.14. The gain from implementing similar parallelism as for the Doppler processing would consequently be next to redundant if used for the pulse compression.

The Doppler processing is where the major part of computing is performed. The trace from the Doppler processing executed in hardware is illustrated in Figure 6.13. We can see that the PS setup time for data transfer is small compared to the total execution time for this function. As mentioned before, in section 6.4.2, some of the Doppler processing are run in parallel in the PS, but this is not visible in the trace collected from SDSoC. Overall, the Doppler processing is a computationally heavy function and the code used in this project was not completely optimized for hardware. The focus of this project was not to optimize the DSP functions to the full extent, but to illustrate how co-design can be used to optimize existing functions in a system. There is a Xilinx IP block available for FFT that would have been faster

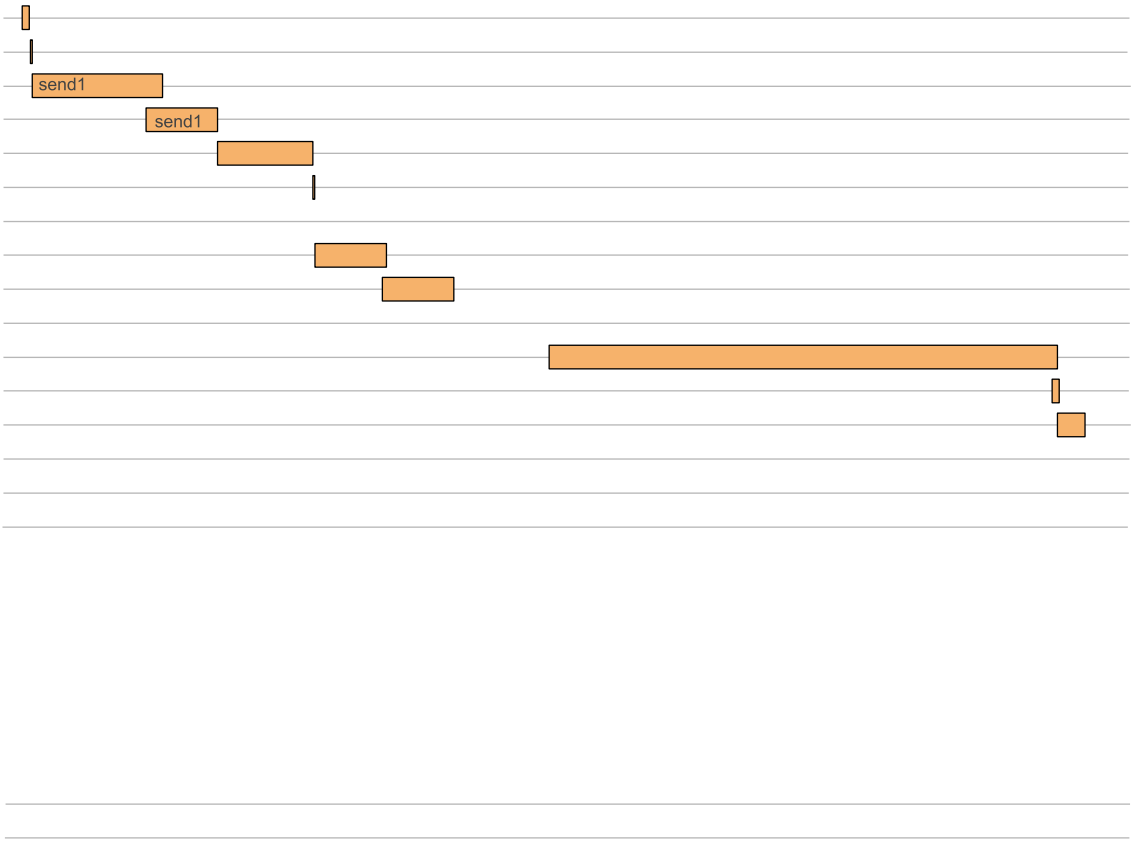

Figure 6.12: Trace of the pulse compression performed in hardware using SDSoC. Parallelism is implemented by using two top functions.

than the one in our design, but those IP blocks would not have been possible to run in PS and co-design could not have been illustrated.

The trace for the complete DSP system is illustrated in Figure 6.14. Important to notice is that only the functionality placed in the PL could be traced using SDSoC. The rest of the functionality, executed in the PS, is added in the figure as grey boxes with lengths concluded from measurements. All data transfers between the PS and PL were done using an FIFO interface. The non-coherent high performance ports AXI_HP0-3 were used together with AXIDMA_sg data movers. Ideally, the Doppler processing should be ready to execute as soon as the pulse compression is finished. The problem is that it is not possible to merge the matrix flipping together with the FIFO interface in the PL. This is due to FIFO requiring a value to be ready every clock cycle, but since the function picks every 189th value and relocates them in a new matrix FIFO cannot be used. For this reason the matrix flipping had to be performed as software, which is why the components could not be pipelined with each other.

Figure 6.13: Trace of the Doppler processing performed in hardware using SDSoC.

The final resource utilization for the DSP system and the individual components are presented in Table 6.5. The system is compared against a full hardware implementation with no co-design for the Doppler processing unit. Notice that the numbers from the individual components do not add up to the total resource usage. The remaining resource usage is due to to multiple different components that the tool needs to synthesize for the design to work, such as external registers and the AXI interface for moving data. The resource usage is slightly less for the co-design which is due to less data having to be moved to the PL, as approximately a third ($\frac{1}{3.2}$) of the computations in the Doppler processing are performed in the PS instead.

Table 6.5: Percentages of FPGA resource utilization for individual components and the final system.

Component	DSP	LUT	LUTRAM	FF	BRAM
Pulse compression	1.4 %	1.3 %	1.1 %	0.5 %	0.2 %
Doppler processing	13.7 %	6.3 %	0.28 %	1.8 %	0.6 %
DSP system (All HW)	15.1 %	22.2 %	3.2 %	14.0 %	10.4 %
DSP system (Co-design)	15.1 %	20.2 %	3.2 %	12.4 %	10.4 %

By using the SDSoC event-tracing functionality and measuring CPU cycles during run-time we established the final speedup of the complete DSP system. The total speedups for the DSP system and for the individual components using a full hardware design and co-design versus running the whole system as pure software are presented in Table 6.6. The highest speedup was for the pulse compression accelerated in the PL, which gained a speedup of 195.8 compared to when executed in the PS. This was an expected number since the FIR filters used for pulse compression

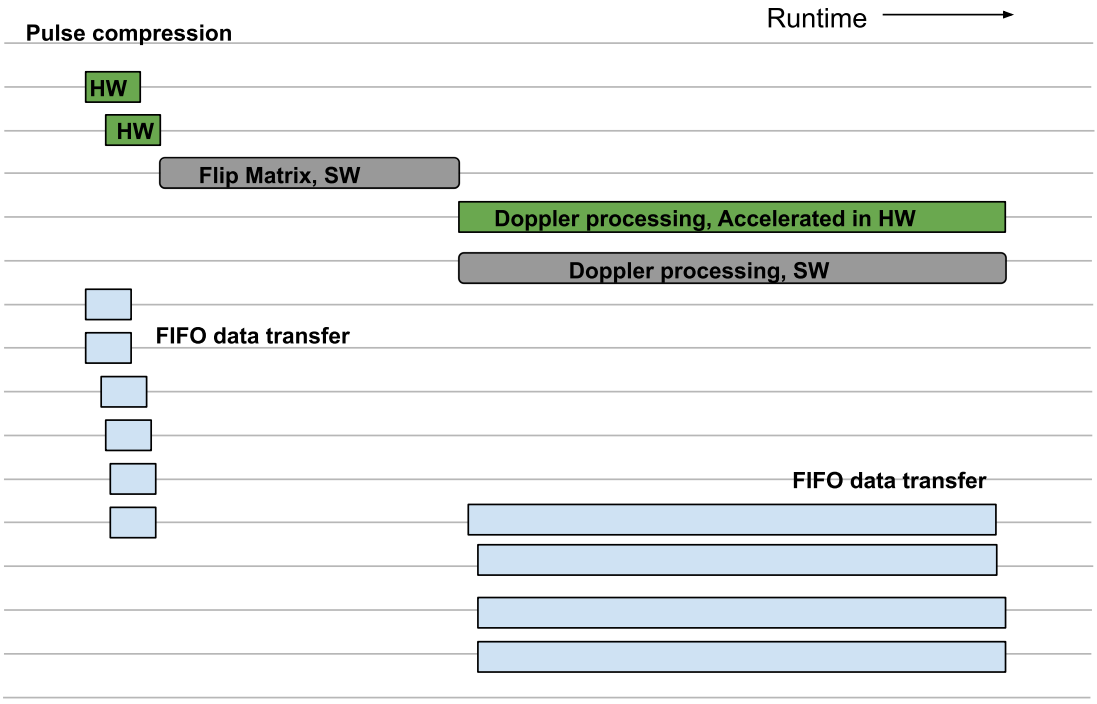

Figure 6.14: Full system trace of complete DSP system. The PS data setup time has been removed for this illustration and boxes representing processing in software have been added.

were optimized for hardware using pragmas and a design that was simple to pipeline. The Doppler processing placed only in hardware gained a speedup of 2.2. However, the speedup improved to 3.0 when using the PS/PL co-design method described in section 6.4.2. Overall, the complete DSP system was 20.5 times faster when both pulse compression and Doppler processing were performed in hardware compared to software. For the final solution, when the Doppler processing co-design was added, the total speedup increased to 23.4. In this project we only investigated the use of the APU in the PS, but if we would add the use of the two remaining cores, the GPU and RPU, the system could probably be accelerated even further. To summarize this investigation, we can conclude that by using a co-design for the DSP system, resources can be saved while also gaining a substantial speedup.

Table 6.6: Speedup for individual components, final system with HW implementation and HW+SW co-design compared to running the system as software only.

Component	HW speedup	HW+SW speedup
Pulse compression	195.8	-
Doppler processing	2.2	3.0
DSP system	20.5	23.4

6.5 Engineering efficiency estimation

Regarding the engineering efficiency for SDSoC, it is interesting to estimate how much the total development time was reduced for the design step where Vivado HLS was automatically used by SDSoC to generate RTL compared to if this step would have been performed manually. We have tried to estimate this speedup, or the increase in engineering efficiency, by comparing the development time for the functions presented in section 6.4 when generated by Vivado HLS compared to when manually developed in HDL. For this estimation, we assume that all HDL components are created from scratch, without using any IP blocks. We also assume that all functionality of the design has been tested and verified using MATLAB before implementation.

To start with, after the initial design idea has been tested using MATLAB, the actual functionality for the MPsoC has to be implemented for either software or hardware. Optimized HDL code must be produced to be able to accelerate a function in hardware, which can be a time consuming process. We estimate that an HLS tool such as Vivado HLS, which is used by SDSoC, could speed up the total development time by approximately 50 % as mentioned in section 4.1.1. This estimation includes several factors such as design time, testing and verification and spin-around time. This of course varies depending on what HLS tool is used, but for the purpose of this investigation we had to make an estimated guess.

Regarding design time, in HDL all kind of optimization in terms of pipelining and timings have to be manually considered, while with HLS this could be automatically optimized. The main drawback would be giving up potential performance that could only be achieved with a manual HDL design. Additionally, during the development of a function or component there are typically lots of iterations together with both testing and verification before it can be released. The main advantage during this stage is that a design made with SDSoC or Vivado HLS could be quickly tested with a C testbench to be sure that the functionality is correct. As opposed to manually creating an HDL testbench and verifying the design with an RTL verification and simulation tool for every iteration this could save a lot of time. As an RTL simulation could take hours for a complex design to finish for every iteration of the design, this could be speed up to minutes with a C testbench with for example GCC. The RTL design after HLS obviously has to be verified with RTL simulation nonetheless but perhaps not for every iteration. These kinds of tools are typically limited in the number of licenses acquired since they are very expensive which also makes it more suitable to use for example GCC as it is free.

In terms of spin-around time, the time it takes to reiterate a design due to possible changes, HLS can also save a lot of time. Changes that have to be done late in the development process might be very complex to do in an HDL design and take weeks to fix. With HLS in SDSoC this could possibly be done by minor changes that instead might take days to correct. The spin-around time it would take to move or accelerate a function in hardware, that was originally designed for software,

would also be a lot simpler. This is because it only has to be changed to be compatible for hardware acceleration instead of having to be redesigned completely in HDL.

Table 6.7: An estimation of the total development time expressed in working weeks, including testing and verification, for simplified versions of pulse compression and Doppler processing.

Component	**Estimated HLS development time**	**Estimated HDL development time**
Pulse compression	1 week	2 - 3 weeks
Doppler processing	2 weeks	4 - 6 weeks

In Table 6.7 an estimation of the total development time when using Vivado HLS compared to when manually programming HDL is presented. The numbers are based on our own experiences during this project together with information concluded from an personal interview with employees at SAAB, Christian Takvam and Anders Bergåker, in 2019 at the department for programmable logic and HFF software for airborne radar. We estimated that the Vivado HLS implementation of an FIR filter used in pulse compression would take up to one working week, including all testing, optimization and verification. The same number by using HDL was estimated as at least two to three working weeks for all the reasons listed above in this section.

For the Doppler processing, we spent two days to develop a Cooley Tukey FFT algorithm in C++ that was synthesizable with Vivado HLS. We did not however spend additional time on optimizing the function for hardware, which would have taken additional time. To make the Doppler processing from initial implementation in MATLAB to a finished optimized and synthesizable design in C++, our guess is that two working weeks would be sufficient. For the HDL implementation, Christian Takvam and Anders Bergåker stressed that development time would be considerably longer, especially regarding testing and verification. They estimated that it would take approximately 4-6 working weeks to do the same kind of implementation. If changes would have been necessary to make late in the design stage, they estimated that the gain from using an HLS tool would be even more apparent. To summarize the estimation, we conclude that the initial guess at a 50 % speedup from section 4.1.1 seems reasonable, but that this number can increase even more if repartitioning of functions is necessary late in the design process.

6.6 SDSoC Evaluation summary

During the investigation of SDSoC in section 6.4 and 6.5 we found that SDSoC can be used to optimize a DSP system using hardware/software co-design. The finished DSP system gained a total speedup of 23.4 when co-design was used to accelerate parts of the design in hardware compared to when the system was only implemented in software. By using optimization pragmas we could alter settings

such as data access pattern, level of pipelining in hardware and the synchronization between hardware and software. Overall we concluded that SDSoC provided a high level of effectively for PS/PL co-design, but that there still are some disadvantages to using the tool that the programmer have to be aware of.

One of the drawbacks with the co-design was that the level of parallelism could be limited for accelerated functions with a large data transfer setup time compared to actual execution time. It is important to be aware that hardware acceleration does not always result in a total speedup for a function, partly for this reason. Another drawback with SDSoC was the limitations using different clock domains in the PL. There are a limited number of clock frequencies that can be used in the PL for an accelerated function. Additionally, when using FIFO for all accelerated functions the system is limited to using the same clock domain as its slowest component.

Overall SDSoC performed well in terms of engineering efficiency. We estimated that by using HLS for the accelerated functions in SDSoC the total design time for the accelerated functions were decreased by approximately 50 %. It was easy to select and move functions that were to be accelerated in hardware as long as the functions were following the limitations for Vivado HLS. The data transfer time could be limiting for some systems but SDSoC still had several options to select level of performance and coherency used for speeding up the transfer. By using the trace functionality we could get an overview of the functionality in the PL, which was useful when making design decisions about parallelism and data transfer settings.

From this investigation we can establish that SDSoC can be used as a complete co-design tool for MPSoC programming. This is done according to the third method in section 4.2.2, where three different methods for MPSoC programming were presented. SDSoC can be used independently and does not require any more tools used for programming. We found that this method was a solid option that made it possible to make decisions about hardware/software partitioning late in the design process.

7
Discussion

In this project we investigated the programming alternatives for a Zynq Ultrascale+ MPSoC by investigating appropriate state-of-the-art tools available on the market today. The results from the investigation were meant to be compared with the current design methodology used by SAAB for MPSoC programming, where a lot of the HDL programming was done manually. We divided this investigation into two different parts, one where we focused only on the RTL conversion for the PL and one where the co-design was further investigated using SDSoC for all parts of the design-chain.

From our investigation we found that Vivado HLS was the optimal tool of the ones investigated for RTL generation for PL programming. We concluded that the tool can be used effectively to speed up the RTL implementation and consequently provide a way for doing the hardware/software partitioning late in the design process. Worth to mention is that although the investigation started with a theoretical prestudy to determine what tools to investigate further, the actual practical investigation only covered a few of the promising RTL conversion tools available. There were some additional options mentioned in the theory chapter that could have been interesting to examine. We were only able to test tools available for SAAB, which limited us to open source, trial versions or tools that SAAB had already purchased. However, Vivado HLS was still the most promising option since SAAB already had expressed interest in investigating the tool as it is optimized for platforms manufactured by Xilinx. The other tools were selected partly because of the theoretical prestudy and partly because of their starting point languages, Python and MATLAB. These programming languages were promising in terms of user-friendliness to work with for SAAB's account. However, the actual investigation performed on the Xilinx MPSoC was not completely fair, since Vivado HLS was the only tool especially designed for Xilinx products while the other tools were merely compatible with Xilinx products. A more accurate investigation could have been made by using other commercial tools specially designed for the hardware used during testing.

Additionally, regarding Vivado HLS, we did not further investigate how to do the actual connection between the PL and the other cores on the board. At the time this project was made, SAAB's current design methodology was to manually program HDL for the PL and HLL for the PS parts of a system. We therefore assumed that there already was a framework available for connecting the PS to the PL, probably by using a design tool such as Xilinx SDK. Regarding SAAB, another benefit with Vivado HLS would be that the tool does not change the current methodology

significantly. The tool can simply replace the current process where HDL is coded manually, without SAAB having to change anything else in their current design methodology. This can be a huge advantage since SAAB is a large company where many systems have to work together and drastic changes in the system assembly is not always realistic.

The second investigation performed in this project was about SDSoC used as co-design method for MPSoC programming. We successfully implemented an DSP system containing pulse compression and Doppler processing by using a combination of the PS and PL. In our investigation, while testing the system, the input data was read from a text file on an SD card. It could have been more interesting to investigate how to stream input data directly from another source such as an A/D converter which would be more realistic in for a system in practise. Parts of the complete system were properly pipelined for hardware, but due to the FIFO data transfer interface between the PS and the PL we were not able to pipeline all the functionality completely. It would be interesting to further investigate how multiple input data packages could be pipelined through the complete DSP system to further optimize the throughput.

Using SDSoC was the only method mentioned in this project that was completely implemented for testing PS/PL co-design on chip. In the beginning of this project, the expectation was that the RTL conversion tool would play a larger role in implementing co-design. We thought that the only way to implement co-design was to use an RTL conversion tool in combination with other independent design tools such as Xilinx SDK. However, SDSoC was discovered during research and proved to be a reliable and promising option. We acknowledge that a more relevant comparison for this project would have been to investigate the Vivado HLS and SDK tool-chain compared to SDSoC instead of investigating several RTL generation tools. Fortunately for this project, Vivado HLS is used as a part of SDSoC and is highly relevant to be familiar with when using the SDSoC for co-design. Additionally, SDSoC requires all source code to a system to be available in the program to be implemented on chip. This can be limiting for a large company such as SAAB, since it would be problematic to move all current functionality to SDSoC. Instead it would probably be most efficient to use SDSoC for testing parts of a larger system, and then implement the changes by using Vivado HLS and exporting an RTL implementation of accelerated functions in the form of IP blocks.

Regarding the engineering efficiency for SDSoC, we have investigated the design step where Vivado HLS was automatically used by SDSoC to generate RTL. Overall, engineering efficiency is a vague expression that is hard to measure. In this project we defined the engineering efficiency as the speedup time gained during development. We found that our initial assumption of a 50 % speedup of the development time was reasonable. However, we were unable to find multiple sources to confirm this number besides the interview with employees at SAAB. It would have been interesting to find a specific study regarding engineering efficiency for Vivado HLS to back up our conclusion. Additionally, we were not able to fairly compare the engineering

efficiency between the tools in chapter 5. Since we had uneven knowledge of the investigated tools when we started the investigation, we were not able to do a accurate comparison of the total development time for only an FIR filter. To perform a more fair comparison an extended investigation of a larger design would have been preferred.

8

Conclusion

It is clear that by utilizing the full power of an MPSoC's processing power, radar algorithms for signal processing can be executed faster due to a higher level of parallelism. Two methods for performing a co-design on the MPSoC have been introduced and compared against the methodology that is used by SAAB today in terms of efficiency and user-friendliness. We have shown that Vivado HLS most likely is a better alternative in terms of engineering efficiency, with a decrease in total development time of approximately 50 %, to realize functions as RTL to be executed on an FPGA instead of manually designing the RTL using HDL. To perform a co-design with this method another tool to develop software and perform the communication between PS and PL must be included. Such a tool was however not part of our thesis project.

We have shown that a complete co-design between the PS and PL of an MPSoC can be made by using the Xilinx software SDSoC, which lets the programmer move a function to be executed as hardware by the click of a button, thus fulfilling the wish from SAAB to be able to move a function as late as possible in the design process. With SDSoC the communication or data motion between the PS and PL is handled automatically while letting the programmer control and change the settings for the data motion network that suits the design. With SDSoC we constructed a co-designed DSP system and tested it on a Zynq Ultrascale+ MPSoC platform from Xilinx. This resulted in a measured speedup of execution time and increased throughput which could be acquired with this methodology.

We recommend SAAB to use Vivado HLS instead of manually designing RTL to realize functions in PL when constructing independent blocks for signal processing purposes. In terms of performing a system co-design, if Vivado HLS is to be used it must be in cooperation with other tools. SDSoC is however very efficient for co-design and has the advantage of being used as a stand-alone tool in the design process. A possible drawback with SDSoC is that needs to embed an entire system into the MPSoC. As it might not be realistic to move large existing systems into SDSoC, this would be problematic for SAAB. SDSoC would however be excellent for testing, tracing and simulate performance of smaller systems or individual hardware and software functions.

Bibliography

[1] "Hardware resource utilization optimization in FPGA-based heterogeneous mpsoc architectures," *Microprocessors and Microsystems*, vol. 39, no. 8.

[2] I. Bahri, L. Idkhajine, E. Monmasson, and M. Benkhelifa, "Optimal hardware/software partitioning of a system on chip FPGA-based sensorless ac drive current controller," *Mathematics and Computers in Simulation*, vol. 90, pp. 145 – 161, 2013, eLECTRIMACS 2011- PART I.

[3] Xilinx Inc., "Xilinx UltraScale ™ MPSoC Architecture," 2014. [Online]. Available: https://www.xilinx.com/publications/prod_mktg/ultrascale-mpsoc-architecture-bacgrounder.pdf

[4] M. A. Richards, *Fundamentals of Radar Signal Processing*, 2nd ed. McGraw-Hill Education, 2014.

[5] M. A. Richards, J. A. Scheer, and W. A. Holm, Eds., *Principles of Modern Radar: Basic principles*. SciTech Publishing, 2010. [Online]. Available: https://digital-library.theiet.org/content/books/ra/sbra021e

[6] Xilinx Inc., "UltraScale Architecture and Product Data Sheet: Overview," 2019, DS890, (v3.9).

[7] Xilinx Inc., "Zynq UltraScale+ Device, Technical Reference Manual," 2019, UG1085, (v1.9).

[8] Xilinx Inc., "Zynq UltraScale+ MPSoC Embedded Design Methodology Guide," 2017, UG1228, (v1.0).

[9] Xilinx Inc., "ZCU102 Evaluation Board User Guide," 2019, UG1182, (v1.5).

[10] Xilinx Inc., "UltraScale Architecture and Product Data Sheet: Overview," 2019, DS890, (v3.8).

[11] SDAccel Development Environment Help, "Understanding FPGA Architecture." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/odz1504034293215.html

[12] Xilinx Inc., "SDSoC Profiling and Optimization Guide," 2019, UG1235, (v2018.3).

[13] D. Koch, F. Hannig, and D. Ziener, "FPGAs for Software Programmers." Springer International Publishing Switzerland, 2016.

[14] Mentor Graphics, "NVIDIA Case Study on High-Level Synthesis (HLS)." [Online]. Available: https://www.mentor.com/hls-lp/success/nvida-hls

[15] Department of Computer Science and Engineering, Indian Institute of Technology Guwahati , "NPTEL - VLSI Design Verification and Test." [Online]. Available: https://nptel.ac.in/courses/106103016/4

[16] Xilinx Inc., "Vivado High-Level Synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[17] MathWorks, "HDL Coder MATLAB Simulink." [Online]. Available: https://se.mathworks.com/products/hdl-coder.html
[18] "MyHDL." [Online]. Available: http://www.myhdl.org/
[19] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.
[20] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*, J. Swiątek, A. Grzech, P. Swiątek, and J. M. Tomczak, Eds. Springer International Publishing, 2014, pp. 483–492.
[21] Xilinx Inc., "Xilinx Software Development Kit (XSDK)." [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/sdk.html
[22] Xilinx Inc., "SDSoC Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html#overview
[23] Xilinx Inc., "Vivado Design Suite." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[24] MathWorks, "Linear Frequency Modulated Pulse Waveforms." [Online]. Available: https://se.mathworks.com/help/phased/ug/linear-frequency-modulated-pulse-waveforms.html
[25] B. R. Mahafza, *Radar Signal Analysis and Processing Using MATLAB*, 1st ed. New York: Chapman and Hall/CRC, 2010.
[26] G. Baguma, "High level synthesis of FPGA-based digital filters," *Uppsala universitet*, 2014, department of Information Technology.
[27] K. Georgopoulos, G. Chrysos, P. Malakonakis, A. Nikitakis, N. Tampouratzis, A. Dollas, D. Pnevmatikatos, and Y. Papaefstathiou, "An evaluation of Vivado HLS for efficient system design," in *2016 International Symposium ELMAR*, Sep. 2016, pp. 195–199.
[28] J. Decaluwe, "MyHDL manual," 2018. [Online]. Available: https://media.readthedocs.org/pdf/myhdl/latest/myhdl.pdf
[29] Xilinx Inc., "Vivado Design Suite User Guide," 2018, UG902, (v2017.4).
[30] Xilinx Inc., "Vivado HLS Optimization Methodology Guide," 2018, UG1270, (v2018.1).
[31] Xilinx Inc., "FIR Compiler." [Online]. Available: https://www.xilinx.com/products/intellectual-property/fir_compiler.html
[32] T. K. Rawat, *Digital Signal Processing*. Oxford University Press, 2015, ch. 11.8.3 Hann (or Hanning) Window.
[33] Xilinx Inc., "SDx Pragma Reference Guide," 2017, UG1253, (v2017.1).
[34] Xilinx Inc., "SDSoC Environment User Guide," 2019, UG1027, (v2018.3).
[35] Xilinx Inc., "SDSoC Environment Debugging Guide," 2018, UG1282, (v2018.2).
[36] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.