A Digital Design Flow - From Concept to RTL Description, Using Mathworks and Cadence's Tools

JORGE DEZA CONCORI & TOR HAMMARBÄCK

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

A Digital Design Flow - From Concept to RTL Description, Using Mathworks and Cadence's Tools

Jorge Deza Concori - `jo8252de-s`
Tor Hammarbäck - `to3567ha-s`

Department of Electrical and Information Technology
Lund University

Supervisor: Steffen Malkowsky
(EIT)

Company supervisors:
Per-Olof Brandt & Joakim Axmon
(BeammWave)

Examiner: Erik Larsson

August 18, 2022

© 2022
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This report presents our digital design flow for creating high speed very large scale integration circuits using a fifth generation disruptive beamforming control and data processing circuit as example. The flow consists of different stages. First, a design stage where a golden reference is created in Mathworks' Simulink using different HDL supported toolboxes like the DSP HDL toolbox. Testbenches are created along side the golden reference in Mathworks' environment with Matlab and Simulink. The same testbenches will be used during the entirety of the flow to continuously verify all of the components in each stage. The second stage is the code generation which will be done using another toolbox from Mathworks, the HDL Coder. To verify the code, Cadence's Xcelium will be used together with Mathworks' HDL Verification toolbox to simulate the produced code using the mentioned testbenches, giving the simulation the same stimuli as used for the golden reference. To automate the flow, automation scripts are created to configure, set up and start the different steps of the stages and connecting the Cadence tools to the Mathworks environment, leaving only the system designing left for the developer. As a final step, Cadence's Genus is used to synthesize the system and evaluate the designs cost in terms of area and speed with the use of a 65 nm standard cell library.

The HDL Coder generates HDL in a hierarchical, readable and well documented fashion with extended debugging properties. The many settings allow for customization of the generation, both in terms of coding style and in architecture. The quality in terms of design costs lies upon the developer. Even though the many toolboxes with predefined algorithms exists to help the designer, custom control systems do not exist as toolboxes and are not better than its implementation as designed by the creator. Using the Mathworks and Xcelium tools in the same environment allowed to reduce the verification time by reuse of testbenches and automate the design of components for a chip concept like the 5G disruptive digital beamforming circuit for BeammWave.

Popular Science Summary

The upcoming 5G-New Radio Standard will enable cellular communication in the millimeter-Wave (mmWave) frequency bands, mainly at 24-30GHz. The mmWave frequency bands open up for large system bandwidth and Gb/s data rates enabling a lot of new use cases, such as advanced Augmented Reality (AR) and Virtual Reality (VR) applications. However, the high isotropic path loss between the radio transmitter and radio receiver for mmWave makes it necessary to rely on antenna arrays with large number of antenna elements. These antenna arrays overcome the path loss by high directional gain through beamforming which focus energy in a particular direction in order to mitigate the high path loss in the mmWave frequency band.

The first 5G mmWave mobile devices on the market, such as smartphones, will use analog beamforming. The main reason is time to market. Analog beamforming solution is bulky and far from optimized with respect to cost, size and power consumption, and hence the first mmWave devices on the market merely constitute a proof-of-concept solution for cellular communication using mmWave.

For smartphones and IoT devices targeting the mass market, the radio architecture really needs to be optimized from cost and size perspective. The start-up company BeammWave has developed a disruptive digital beamforming architecture, by integrating the RF IC, front end Radio modules, filters, and antenna element in a single RF chip that will drastically change the way beamforming will be implemented in mobile devices in the future. Future mobile devices will rely on digital beamforming.

With increased system complexity, methods and tools that can shorten the design development time become increasingly important, specially concerning digital designs. Traditionally, a system description is typically "manually" translated to VHDL or Verilog and then introduced into a digital flow. Recently, tools have appeared that can automatically convert Matlab/Simulink system descriptions using e.g., the HDL Coder toolbox in Matlab, to generate HDL that can be introduced into the digital flow in Cadence's tools. With more complex systems the capability

of these tools needs to be evaluated.

In this Master's Thesis, we present our flow, how to connect and use the different state of the art tools. We will also show how to automate the processes by using real digital designs needed by BeammWave in their disruptive digital beamforming, as examples. This flow includes everything from the design process to creating idealized models. From generating HDL and verifying its behavior, to synthesizing which estimates the cost, size and timing. Finally the flow is evaluated, not only in the resulting quality compared to a traditional flow, but also from the developing perspective, comparing traditional and our flow in terms of automation and hours spent for development.

List of abbreviations

5G-NR:	5G-New Radio
ADC:	Analog to Digital Converter
ASIC:	Application-specific integrated circuit
CAN:	Controller Area Network
CPU:	Central Processor Unit
DAC:	Digital to Analog Converter
DSP:	Digital Signal Processing
FIR:	Finite Response Filter
FPGA:	Field Programmable Gate-Array
FSM:	Finite State Machine
HDL:	Hardware Description Language
HSDPA:	High Speed Downlink Packet Access
IC:	Integrated Circuit
IoT:	Internet of Things
IP:	Intellectual Property
RF:	Radio Frequency
RFIC:	Radio Frequency Integrated Circuit
RTL:	Register Transfer Level
SoC:	System on Chip
SRAM:	Static Random Access Memory
TCL:	Tool Command Language
VHSIC:	Very High Speed Integrated Circuit
VHDL:	VHSIC Hardware Description Language
VLSI:	Very Large-Scale Integration

Table of Contents

1 Introduction 1

1.1 Motivation . . . 2

1.2 Goal of the Project . . . 2

1.3 Related Work . . . 2

1.4 Thesis Outline . . . 3

2 Overview 5

2.1 Digital Design Flow . . . 5

2.2 General VLSI Digital IC Design . . . 6

2.3 Hierarchical Design . . . 7

2.4 Our Design Flow . . . 9

3 Idea and Modeling 11

3.1 Product Idea . . . 11

3.1.1 Mathworks Model 11

3.1.2 Testbenches 12

3.2 Model Development by Examples . . . 13

3.2.1 Edge Detection - A 2D Convolution Algorithm 13

3.2.2 DIG-IF 15

4 HDL Code Generation 17

4.1 Fixed Point . . . 17

4.2 HDL Coder . . . 19

4.2.1 Code Settings 20

4.2.2 Blocksets 22

4.2.3 Dedicated Blocks 23

4.3 Handwritten HDL Development - Edge Detector . . . 24

5 Simulation and Synthesis 27

5.1 HDL Verifier . . . 27

5.2 Cadence's Xcelium . . . 28

5.3 Xcelium Co-simulation . . . 30

5.4 Synthesis using Genus . . . 32

6 Results 33
6.1 Models . . . 34
6.1.1 2D Convolution 34
6.1.2 DIG-IF 39
6.2 Manual translation vs HDL Coder generation . . . 45
6.2.1 HDL Coder Generated 45
6.2.2 Manual Coding 45
6.3 Genus Synthesis . . . 47
6.3.1 Coefficient Banks 47
6.3.2 DIG-IF Requesting 48
6.3.3 Combining Block 48
6.3.4 Handwritten Edge Detector VS Generated Edge Detector 49

7 Conclusions 51
7.1 Comparison . . . 51
7.1.1 Quality 51
7.1.2 Developer Experience 52
7.2 Setup . . . 52
7.3 Future Work . . . 53
7.3.1 Improving the Flow 53
7.3.2 Extending the Flow 53

References 55

A DIG-IF 57
A.1 Design Specifics . . . 57
A.1.1 Requesting 57
A.1.2 Responding 59
A.2 Constraints . . . 60
A.2.1 Constants 60

B Matlab code 61
B.1 Github repository . . . 61
B.2 Matlab test suite - Script to test all the components . . . 61
B.3 Matlab setup constants script . . . 64
B.4 Matlab generate test data . . . 66
B.5 Cosimulation script in matlab . . . 69

C VHDL code 71
C.1 HDL Coder RAM block . . . 71

D TCL scripts for synthesis 73
D.1 synth.tcl . . . 73
D.2 Design_setup.tcl . . . 74
D.3 create_clock.tcl . . . 75

List of Figures

2.1 Gajski–Kuhn chart or Y diagram 5
2.2 Design flow steps 6
2.3 Hierarchical structure in Simulink of DIG-IF 8
2.4 Proposed digital flow 10
2.5 Flow components 10

3.1 DIG-IF Design sketch 15

4.1 Fixed point with 3 fractional bits 18
4.2 Fixed point with slope $1.5 \cdot 2^{-3}$ and bias $+1$ 18
4.3 fixed point tool data types 19
4.4 HDL Coder Wizard 20
4.5 HDL Coder output reports 21
4.6 Simulink register 22
4.7 8 bit binary index search 23
4.8 HDL Coder Dual port RAM in Simulink 23
4.9 Timing diagram from HDL simulator 25

5.1 Blocks in the HDL Verifier blockset 27
5.2 Connection Setup in co-simulation block 28
5.3 Cadence Xcelium environment 29
5.4 Simvision GUI 29
5.5 Xcelium as a server 30
5.6 Design model and co-simulation linking model 31
5.7 Schematic view from Genus 32

6.1 Overview of final flow 33
6.2 The flow's file structure 34
6.3 The 2D convolution result 35
6.4 Original black and white, 8 bit image 36
6.5 Outer model of 2D convolution implemented in Simulink 38
6.6 Outer subsystem of 2D convolution in Simulink 38
6.7 Simulink and Xcelium simulation differences 39
6.8 Zoomed in Simulink and Xcelium simulation differences 39

6.9 DIG-IF subsystems 40
6.10 Request output ID signal 42
6.11 fixed point filter results 43
6.12 combiner input 44
6.13 combiner results 45
6.14 Cosimulation waveform of handwritten edge detector 46
6.15 Co-simulation on Handwritten HDL 47

List of Tables

6.1 Incoming requests 41
6.2 Part of the request output 41
6.3 Coefficient Banks RAM gates and area synthesis report 47
6.4 Resources used by coefficient bank control module 48
6.5 Resource used by combining block 48
6.6 Results from handwritten and generated Edge Detector 49

A.1 Request data parameters 57
A.2 Configuration request parameters. 59
A.3 Constants 60
A.4 Attributes 60

Chapter 1

Introduction

New improvements to smartphones, computers, internet of things (IoT) and other devices, both in processing and data usage, creates more and more complex systems. The process of developing new digital integrated circuits (IC) has evolved from simple units to complex systems which has driven the development and verification time to increase considerably. Often large teams divide the work in different abstraction layers and subsystems. From idea and mathematical representation of the subsystems down to register transfer level (RTL) and transistor level design.

While the complexity has increased, new tools to handle the complexity have been developed. Mathworks' Matlab and Simulink have added or updated toolboxes to enable translation of their code to hardware description languages (HDL) which describes the RTL. Integration of verification tools such as Cadence's Xcelium or Mentor graphics' Modelsim have also been added to Mathworks' tools.

One complex system is the radio frequency (RF) IC for the 5G radio standard which the start-up company BeammWave currently is developing. One part of the RF IC is the digital processing and control system of the down and up link, between the antennas and the 5G-NR (Fifth Generation New Radio) baseband circuitry. This design will be implemented to explore and create a flow around since BeammWave is kind enough to supply concepts and description of the state of the art system.

The main task of this master thesis is to create a design flow for developing digital ICs and evaluating compared to traditional flow using the tools from Mathworks and Cadence. To evaluate the flow we will compare generated HDL from the Mathworks tools with a traditional creation of HDL by analyzing the properties given by synthesizing the RTL descriptions. Evaluating the flow by its development efficiency compared to traditional translation will also be included in this master thesis.

1.1 Motivation

The first 5G mmWave mobile devices on the market, such as smartphones, will use analog beamforming. The main reason is time to market. Analog beamforming solution is bulky and far from optimized with respect to cost, size and power consumption. BeammWave has developed a disruptive digital beamforming architecture, by integrating the front end Radio modules, filters and antenna element in a single RF chip which will change the way beamforming will be implemented in mobile devices in the future.

Traditional verification requires manual translations from system description to RTL which often takes a long time. Especially when large parts of the code have to be redone when something changes. A quicker design flow is developed and to decrease the developing time and thus decrease the time to market which benefits any company.

A design flow using Matlab to create RTL is not new [1] but with the continuously updated and added toolboxes to Matlab and Simulink makes it worth investigating the current state of the tools to create state of the art technology. Specifically since Matlab is often used in the design process by researchers from different fields. From ideas and mathematical descriptions to RTL in a single environment makes the flow centralized and convenient.

1.2 Goal of the Project

A documented design flow for RTL development using Mathwork tools, with clear instructions on how and where to configure in order to meet specifications for any future application specific integrated circuit (ASIC). A big part of the flow will be testing and verification to make the development easier. The flow will be flexible to enable the adding of new blocks, toolboxes and tools to continue the creation of state of the art technology.

The master thesis will also evaluate the flow compared to traditional manual translation flow. This will show advantages and disadvantages between manual and generated HDL and how the developing time differs. The findings of analyzing different verification methods and tools will be presented along with design problems and how to solve them. Finally, a report of the digital design flow principles will be included in the master thesis.

1.3 Related Work

Past studies of system and RTL verification flow have been done in Matlab [2] where O. Z. Batur, M. Koca and G. Dundar created a synthesizable VHDL model using Matlab scripts and testbenches which are simulated using Mentor Graphics' Modelsim as co-simulator, comparing results between the reference model and the RTL model. In another study [3] K. Ng explores different challenges in RTL verification using high-level system models, where the approach is to simulate a reference model and a model under test and compare their results. How effective these models are in particular issues such as timing and input constraints in order to verify the RTL model. Since high-level system models are a reference for the hardware implementation and do not describe entirely the architecture of the system.

Another study which includes the Cadence Xcelium tools are [4] A. Goel, B. B. T. Sundari and S.Mathew's "UVM based Controller Area Network Verification IP (VIP)" a verification intellectual property (IP) is tested for a Controller Area Network (CAN) block. This approach is simulating using Xcelium in order to perform the functional verification. In [5] T. M. Bhatt and D. McCains "Matlab as a development environment for FPGA design" a design flow using Mathworks tools targeted to Field programmable gate arrays (FPGA) presents how a digital flow is applied to High Speed Downlink Packet Access (HSDPA) blocks, by using older tools such as HDL Designer and conversion blocks. The authors create a design flow for RTL development and verification, along with Mentor Graphics software. T. M. Bhatt and D. McCains flow can be improved with more recent tools and targeted to ASIC as it is intended in this master thesis. A more recent study [6] is V. Y. Sarges "Evaluating Simulink HDL Coder as a Framework for Flexible and Modular Hardware Description" which evaluates Mathworks HDL Coder, for specific blocks for a FPGA.

1.4 Thesis Outline

The presented thesis is organized in the following chapters:

- **Chapter 2: Overview.** A summary of the modern digital design flow techniques and our digital flow for this master thesis.
- **Chapter 3: Idea and Modeling.** Describe how the Mathworks environment is used with different tools by exploring designs and forming the first stages of our digital design flow based on an edge detector and DIG-IF models.
- **Chapter 4: HDL Code Generation.** Shows how the Mathworks blocksets are translated to HDL together with the design process of writing HDL by hand.
- **Chapter 5: Simulation and Synthesis.** Describes the used Cadence tools, the strategies to setup and automate the process based on the mentioned models from chapter 3.
- **Chapter 6: Results.** A description of the designed digital flow and the generated HDL code, co-simulation and synthesis tasks based on the models from chapter 3, compared to the handwritten.
- **Chapter 7: Conclusions.** Final conclusions regarding the master thesis and future work for this flow.

Chapter 2

Overview

2.1 Digital Design Flow

A digital design flow is a process for developing a concept or a product with certain specifications to physical hardware implementation. The process is usually seen from three different domains: The behavioral, the structural and the physical. Figure 2.1 shows the Gajski–Kuhn chart levels from the center of the diagram to the outside with a higher level of abstraction.

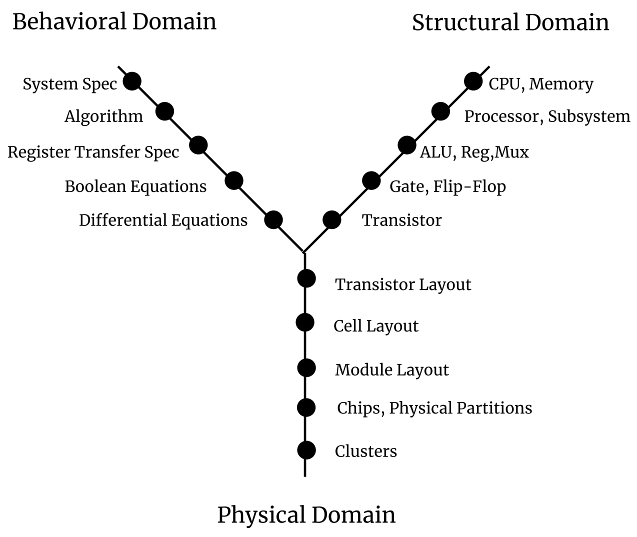


Figure 2.1: Gajski–Kuhn chart or Y diagram

The behavioral domain represents the system as a black box, since it explains how the system responds to stimuli by using equations to describe the functionality. The structural domain is more precise by specifying the actual components inside the hardware implementation and how all these components are connected.

A netlist gathers these links representing it as a schematic. The physical view adds more information to the system such as the placement of each component, including the used area and wires in the system implementation and is used for fabrication.

2.2 General VLSI Digital IC Design

For Very Large Scale Integration (VLSI) fabrication, meaning the fabrication of a system that could include millions of transistors, a typical flow [10] is followed in different stages, as in Figure 2.2.

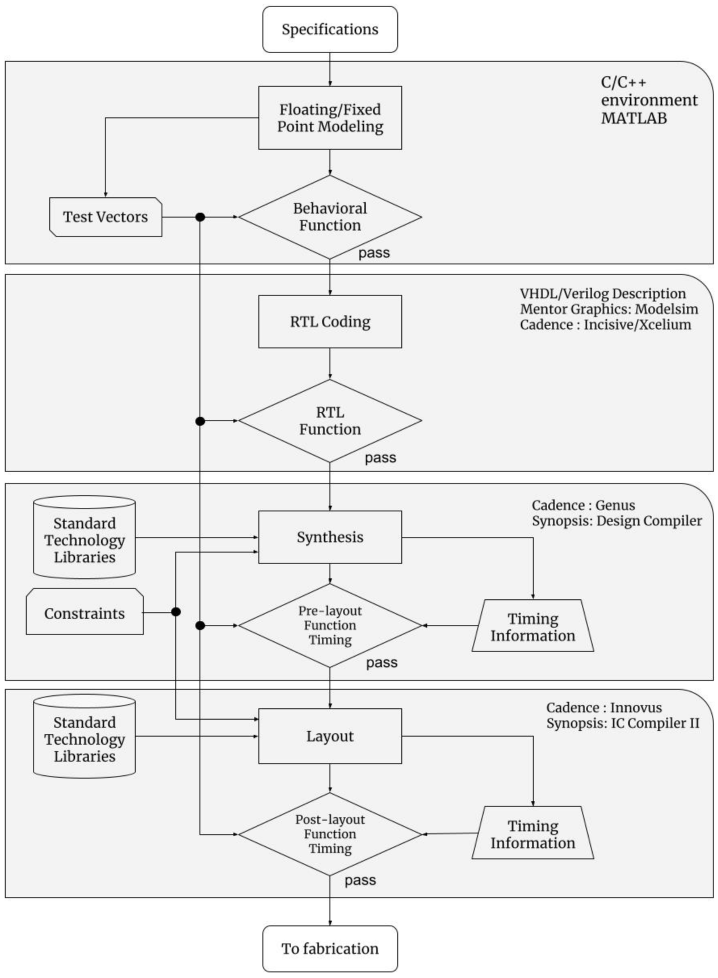


Figure 2.2: Design flow steps

The first stage is to create a floating or fixed point model and verifying its behavior. Most other programming languages are capable of creating and testing models since it often describes a algorithm or function.

The next step in the stage is in the structural domain. Describing the model in RTL using HDL such as VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) or Verilog. The systems are often described using Finite State Machines (FSM) which control the systems behaviour only using states and inputs. At the RTL, all registers, signal widths and logic gates are defined which creates a complete system description. These first two stages is the frontend of VLSI design and the following stages are the backend VLSI design [12].

Synthesis is the third stage where the components are compiled from a system description in RTL. Hardware libraries are used to define how the transistors should be used to create the logic gates and registers. It also contains clocks circuitry which is Synthesized to include timing constraints. The clocks are defined with a frequency, maximum skew and a maximum delay. Since transistor takes time to switch it is crucial that all the transistors have had enough time, in the clock cycle time frame to switch. The synthesising sums up the switching time of each transistor, making sure the timing constraints are met. Synthesizing programs like Cadence's Genus can optimize the system by switching places of components to improve the timing characteristics. But if the timing constraints still are not met, a redesign of the logic is needed or a loosening of the constraints.

So far no physical schematic has been created, everything has just been simulated. Synthesizing just calculates the design as everything where well behaved without parasitic currents. In the last step of the stage, the routing, a physical layout is created. If problems arise in this stage, the redesign may go back to the synthesis or even to the RTL creation stage. A digital design flow is an iterative process [7] where design validation, constraint checks, timing and size verification are performed in parallel, often by different engineers. After all the tests have passed in the all stages, the design can be manufactured to a physical product.

2.3 Hierarchical Design

As a system becomes larger and complex, going through the mentioned stages in a flow becomes harder to modify and validate. Using a hierarchical design flow, the system can be broken down into smaller blocks or subsystems. Each block or subsystem can then be divided into more blocks or subsystems until the description is implementable. The testing is also divided into different types of tests such as unit tests for each block or system and integration tests for testing blocks and systems together.

Figure 2.3 shows how a design can be divided into more blocks in Simulink. By designing in a hierarchical way, advantages such as tests for specific blocks, separate the workload in a team, the reuse of verified designs to similar systems and enabling the integration of third party components into the system [9].

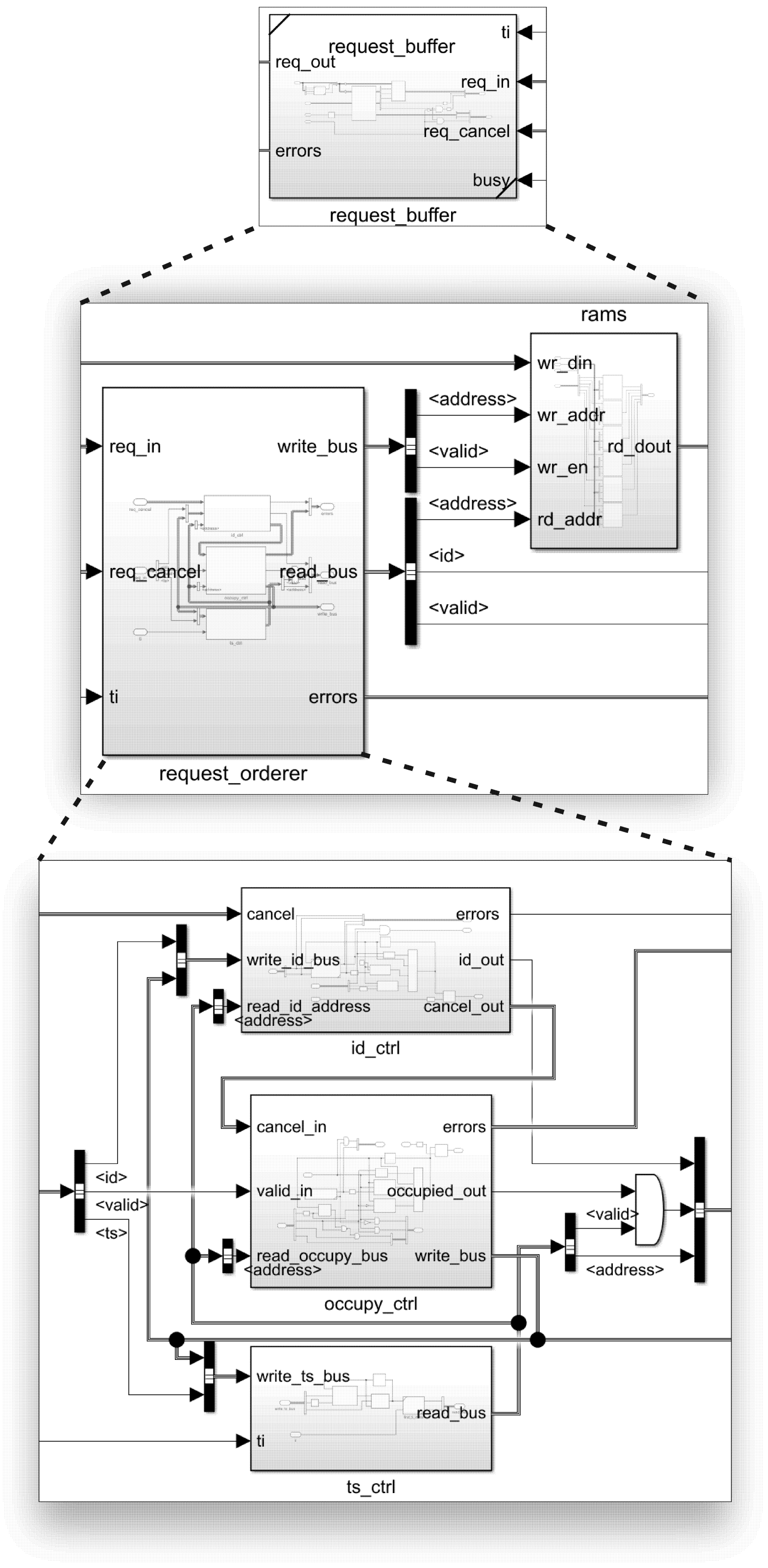


Figure 2.3: Hierarchical structure in Simulink of DIG-IF

2.4 Our Design Flow

After a general digital flow was reviewed and exploring in previous sections, the following chapters will specify with tools, automation scripts and examples how our flow works. How certain packages from each software links to another to create an efficient design flow and how to construct each step in the stage for automated processes. The Figure 2.4 displays the entire digital flow, which shares procedures from a general design flow as shown in Figure 2.2.

The main focus of our flow goes from a concept or a product idea to verification and synthesis of the RTL description. It also focus on how the components are created, both in terms of product quality and the developing efficiency. By using Mathworks' environment, the entire flow can share testbenches, both across different stages and for different components. A network link is used to setup a connection to Cadence's Xcelium and Genus as an extension of the environment, allowing co-simulation for RTL description verification and synthesis. A flow can be created for many different types of components. The components our flow are based on is shown in Figure 2.5 which is picked out for developing control circuits and High Speed (HS) data processing using filters and basic arithmetic from sampled analog signals.

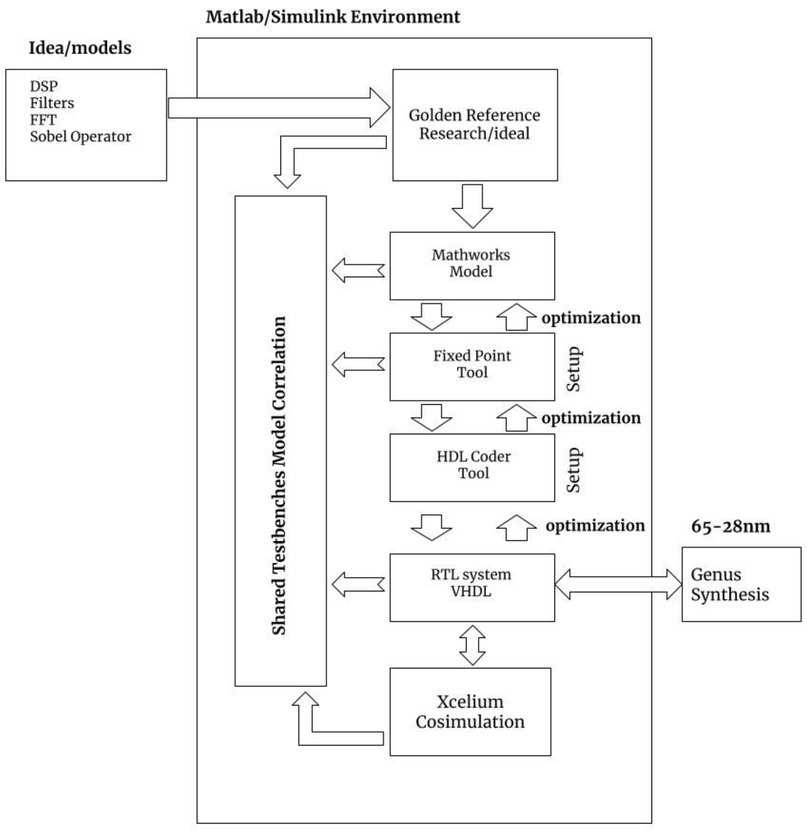


Figure 2.4: Proposed digital flow

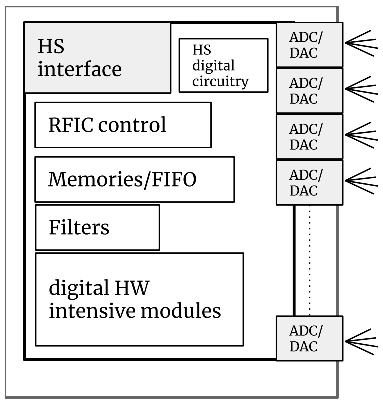


Figure 2.5: Flow components

Chapter 3

Idea and Modeling

3.1 Product Idea

To create an idea of a product in digital design, the development starts with the defining the function and its constraints, these are studied and explored with the Mathworks tools to create a golden reference. It is called in this way, golden reference, due to it determines the target, how the targeted design should work in a idealized way. Then by using different Mathworks toolboxes a digital model can be formed by dividing the design into smaller comprehensible components in a hierarchical way.

Simulink is suitable for this task since it shares the same perspective of hierarchy where each component is modeled by itself and then connected, which is also the structure in RTL system description. In the Mathworks enviroment, scripting can be done to create testbenches and connecting the models to other services provided by toolboxes and other programs. The outputs of those services can be processed and verified with scripts to make sure the targeted design works as intended. The testbenches are solely based on the Matlab programming language and not generated to hardware testbenches as other flows may have. The benefits of having the testbenches outside of the hardware environment is that they also can be used in the later stages design stages, such as the RTL generation and synthesis stages discussed in chapter 4 and 5.

3.1.1 Mathworks Model

From the golden reference, components are made. Each properly defined as its own model by choosing and defining its specific algorithm. The models are created as Simulink blocks which can be translated to HDL. The modeling is a continuous development where the models are updated to fit the constraints of the product since all the cost parameters of the modeling are not known until synthesis or even the final routing stage. This is why each model needs its own testbench and automated setup, to collect the costs as far down the design flow as possible.

To differentiate between what is going to be translated to HDL code and what are constants, test data and testbenches, Simulink is only used for what is going to be translated. Matlab code could also be translated to HDL code but avoided as explained more in chapter 4. Some exceptions are made but the code for the exceptions are put in Simulink's Matlab function blocks which works as any other Simulink model.

To design a circuit in Simulink, different predefined operational blocks are used. The blocks connect to each other by drawing lines between each block's inputs and outputs, similar to a signal in VHDL. Just as signals in VHDL have different types, so does and have the lines. The type define how the blocks should operate on the signal. Array types are also available creating a list of signals. But instead of defining the type for each line, it propagates from the blocks input and output connections. Often the testbench input defines the type of the input signal for the first blocks which output can either change the type or be propagate the type, leaving it the same as the input. For example, an addition block can increase the bitwidth with one bit to avoid overflow, changing the type. A delay block does not change the type and propagates the input type to the output.

3.1.2 Testbenches

Creating the models and testbenches is done in unison, continuously verifying earlier made models with integration testbenches. But before the modeling starts, a setup script is created where test data is generated from the description and constants. The test data is saved and used for every model's testbench. That way the same test data can be used for multiple iterations of the same model, making the testing efficient and centralized. The generated test data is the same input as the top model would receive, meaning it has to be transformed before it is used for each specific model.

The testbench starts by importing the constants and subset of test data needed. Then it transforms the test data to fit the specific model. The easiest way is to use previous models to transform the test data which also integrates those models to an integration testbench. Code 3.1 shows a part of example integration testbench, how it uses a previous model to transform the test data.

```
function errors = integration_testbench(verbose, options)
    load('setup/example_input.mat', 'input_bus');
    load('constants.mat', 'req_mem_size');

    input = [input_bus.req_base];
    stop_time = numel(input);

    % Transform input using previous model
    packager_output = sim('packager/unit_test', [], options.simOptions);

    % Runs the test
    input = packager_output;
    output = sim('request_buffer/unit_test', [], options.simOptions);
```

Code 3.1: Part of Request buffers unit test

After the creation of the specific test data comes the generation of expected output data. This is done either directly from the constants, from the input or as a combination. If the model should be able to handle certain errors, such data must also be generated and tested for. This is done by editing both the test data and the expected data in the specific testbench.

The testbenches are built as functions in Matlab, taking run options and debugging arguments such as verbosity as inputs and returns if there is any mismatch between the received and expected output. The functions are collected in a script as a test suite where all tests can be run with the set of input data and constants. This ensures that every model still works after changing the constants. In addition to the Matlab testbench suite which tests the functionality of the models, it also connects to a co-simulation server as described in chapter 5 to verify Simulink's simulations using the same test data.

3.2 Model Development by Examples

3.2.1 Edge Detection - A 2D Convolution Algorithm

As a first example, a two dimensional (2D) convolution also called a finite impulse response (FIR) filter was chosen to explore the design flow. A FIR filter is often used for image processing and in this example the filter will be used to find edges in an image. It was chosen since the data is streamed through the algorithm and it needs large memories. Both streaming and handling memory are crucial in many hardware designs which makes how the Mathworks tools handle this essential for establishing the flow.

The golden reference starts by examine the mathematical expression of a 2D convolution which looks as follows:

$$g(x, y) = w * f(x, y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} w(dx, dy) f(x + dx, y + dy)$$

where g is the convoluted result, w and f are the inputs to be convoluted. In image processing one of the inputs is called a kernel w which operates on the image f.

To verify that the actual algorithm works, a testbench is set up in Matlab where the input is changeable, both using different images but also with raw numbers for debugging. The output can then be verified by examining the plot of the output or in this case, the processed image. Mathworks have a lot of predefined functions and one of them: `filter2` [15], does the exact 2D convolution. Providing the test image and kernel an expected output can be generated and compared to the actual output, verifying the algorithm with pixel precision.

3.2.2 DIG-IF

As a second example design to create a flow for, a description was set up together with BeammWave for a part of their 5G chip, the DIG-IF. Specifically it describes digital part between the ADCs (analog to digital converters) and an interface for a processor or a state machine, the baseband (BB).

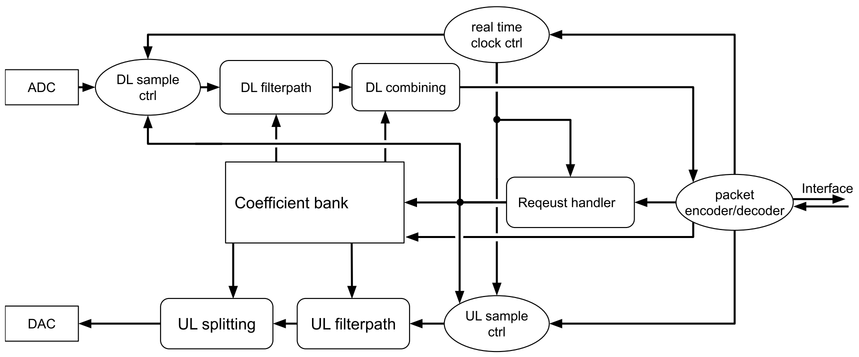


Figure 3.1: DIG-IF Design sketch

The function of the DIG-IF is to control the downlink (DL) data flow between antennas and baseband by filtering and combining the data streams using different filters and combining weights. It also controls the uplink (UL) data flow in a similar way as the downlink but instead of combining the data streams from the antennas, it splits the data stream from the baseband to the antennas.

The DIG-IF design is built using a request scheme. The baseband requests data, the change of a parameter or the setting of a coefficient. The data requests includes which time to sample since the baseband does not keep track of the exact time. Each request makes DIG-IF respond with an acknowledgement or a negative acknowledgement, depending on the success. The DIG-IF will also respond later when it is time to sample the data from a data request. The request are not necessarily sent in order from the baseband and can be canceled, meaning search algorithms are needed to insert and read the requests stacks.

All the design specifics for DIG-IF can be found in appendix A together with the constraints, final circuit and results.

Chapter 4

HDL Code Generation

4.1 Fixed Point

Before converting a Simulink model to HDL code the signals must be represented in integer or fixed point form. Initially the signals are Mathworks' default type: double, defined by IEEE's Standard 754 [8]. The reason the conversion is needed is because fixed point form have a lower complexity leading to using less space, less power and an increase in speed compared to a floating point implementation. The only loss is the dynamic range floating point provides but since the range of a signal often is known, the dynamic range is not needed.

Fixed point form is a signal type with a fixed bitwidth, fractional length and a bit representing if the signal is signed, using two's compliment. It also defines the signal scaling, bias point and how mathematical operations work regarding rounding and adding additional bits to avoid over- and underflow. Operational blocks can be specified if they should operate on the signals using real world signal value with the scaling and biasing or using the stored bits as they are, resulting in different hardware implementations. For fixed point signals with binary point scaling and zero bias, real world and stored bit hardware implementations are the same. Figure 4.1 and 4.2 shows how `0b11010101` can represent different numbers in fixed point form.

Figure 4.1 has three fractional bits meaning the three least significant bits represent the decimal part. The decimal part is represented as division by 2. One fractional bit would only allow the decimal to be 0 or 0.5. Note that it also signed with its most significant bit set to high, this makes the number be negative.

Figure 4.2 is not defined by a fractional point. Instead it is defined with a slope. Each increasing bit is scaled with $1.5 \cdot 2^{-3}$ resulting in the different bits can represent values a binary point scaling can not.

In the edge detection example, the pixels of the image already are defined in a black and white range, represented by 8 unsigned bits and a kernel which only includes integers between -1 and 1 inclusive. The input types propagate through

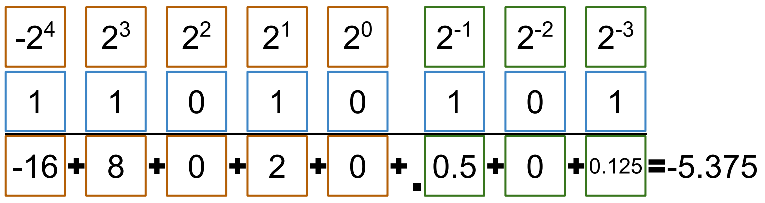


Figure 4.1: Fixed point with 3 fractional bits

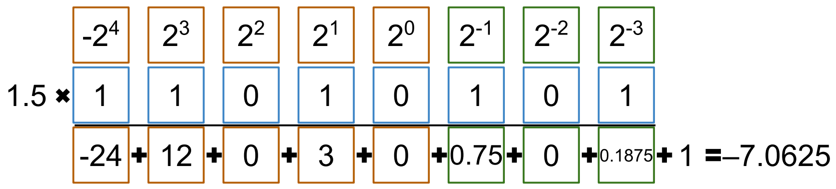


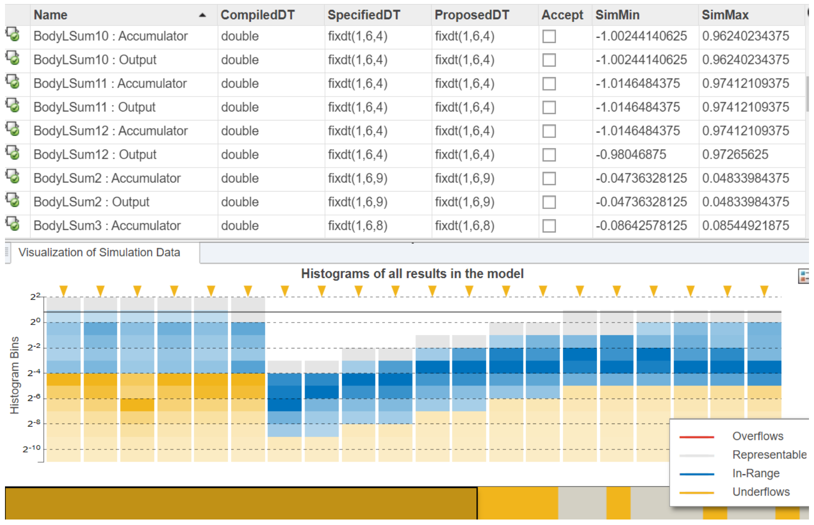
Figure 4.2: Fixed point with slope $1.5 \cdot 2^{-3}$ and bias $+1$

the model which makes the output gain a signed bit from the multiplication and a few bits from the summations, compared to the input.

If the kernel had had fractions with a wider bitwidth the resulting output bitwidth could have been increased to avoid overflow or underflow. But high precision has it costs and can lead to unreasonable high bitwidths. In the edge detection case, an 8 bit input multiplied 9 times in a 3 by 3 kernel with another 8 bit number would result in the output needing to be 72 bits width to avoid overflow. The solution is to specify the type, the fixed point parameters, after each operation and allowing some underflow. Mathworks' provides an tool, the fixed point tool which suggest new parameters since calculating them for every step by hand can be a tedious task.

The fixed point tool works by iteratively trying different fixed point configurations for different signals in the specified design and evaluating the output error for different inputs. When the iteration max is hit or the absolute error is below specified allowed error a report is generated. The report consisting of how much each of the bits are used in each component together with overflow and underflow.

The newly found fixed point configurations can then be implemented in the actual design. Most of the DIG-IF design is a control system which is bit exact and where every bit size is predefined. The only exception is the up and down link data paths where the filters use the fixed point tool. The fixed point tool wizard is shown in Figure 4.3 and the result of a fixed point filter implementation for the constraints of the DIG-IF is shown and discussed in section 6.1.2.3



Name	CompiledDT	SpecifiedDT	ProposedDT	Accept	SimMin	SimMax
BodyLSum10 : Accumulator	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-1.00244140625	0.96240234375
BodyLSum10 : Output	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-1.00244140625	0.96240234375
BodyLSum11 : Accumulator	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-1.0146484375	0.97412109375
BodyLSum11 : Output	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-1.0146484375	0.97412109375
BodyLSum12 : Accumulator	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-1.0146484375	0.97412109375
BodyLSum12 : Output	double	fixdt(1,6,4)	fixdt(1,6,4)	☐	-0.98046875	0.97265625
BodyLSum2 : Accumulator	double	fixdt(1,6,9)	fixdt(1,6,9)	☐	-0.04736328125	0.04833984375
BodyLSum2 : Output	double	fixdt(1,6,9)	fixdt(1,6,9)	☐	-0.04736328125	0.04833984375
BodyLSum3 : Accumulator	double	fixdt(1,6,8)	fixdt(1,6,8)	☐	-0.08642578125	0.08544921875

Figure 4.3: fixed point tool data types

4.2 HDL Coder

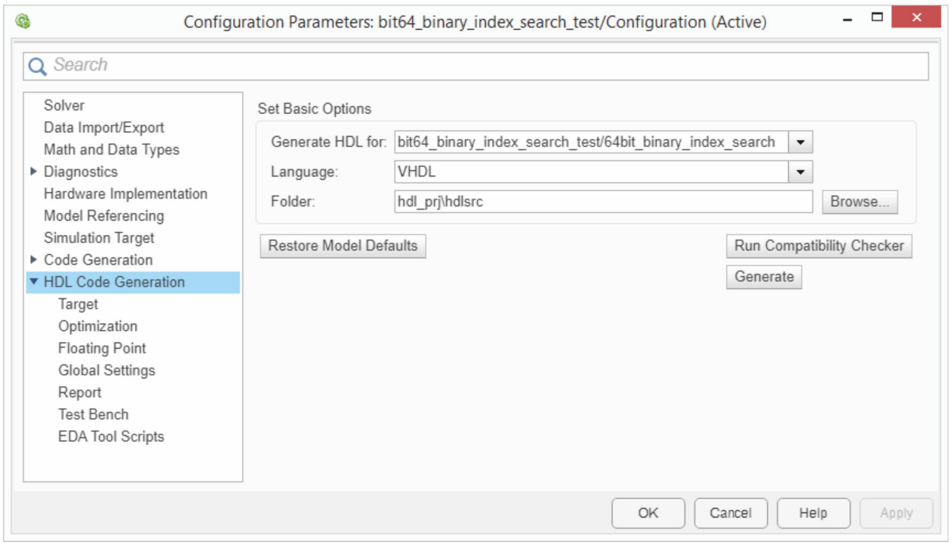
Mathworks' HDL Coder toolbox generates HDL from Matlab functions or Simulink models. The generation works by having predefined HDL code for the fundamental blocks and connect them using the Simulink models or Matlab functions as schematic. This way makes the HDL Coder suitable for RTL hierarchical designs where a complex system can be treated and imported as a group of small subsystems and developed independently. The HDL Coder toolbox provides a library of blocks including bit operations and routing specifications. The toolbox also grants the initial blocks of Simulink such as adders and multipliers, to be HDL compatible, giving them fixed point algorithms for HDL realization. The HDL Coder also provides some larger hardware models such as RAMs, serialization and deserialization systems to tune the design in terms of speed and area.

The HDL Coder can also optimize the implementations HDL by pipelining and make subsystems share resources such as multipliers, adders and other DSP blocks. Neither the 2D Convolution model or the DIG-IF has used resource sharing since no resource limits has been set. Commonly resource sharing is revisited after synthesis where a size and resource usage estimation is concluded. When developing for a FPGA the resource limit is known since a FPGA has a fixed amount of resources and the optimization is of higher importance.

Verification models are also generated by the HDL Coder. These models have all their variables set as fixed values and its generative functions have been generated meaning the model does no longer depend on Matlab workspace variables or have any variant subsystems. In the verification model the optimizations can also be seen. For example, either by the absence of blocks due to dead code and resource sharing or by insertion of extra registers in case of pipelining.

4.2.1 Code Settings

There are two ways to generate code. Either the usage of a window based wizard as displayed in Figure 4.4 or to call a Matlab function with the design parameters as shown in code 4.1.


Configuration Parameters: bit64_binary_index_search_test/Configuration (Active)

Set Basic Options

Generate HDL for: bit64_binary_index_search_test/64bit_binary_index_search

Language: VHDL

Folder: hdl_prj\hdlsrc

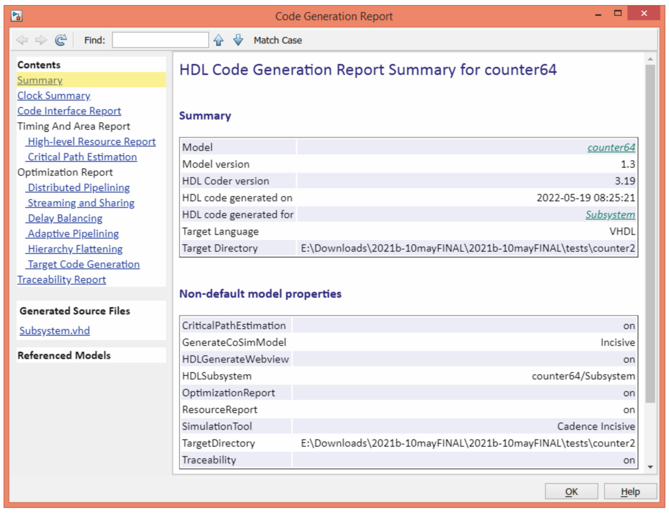
Figure 4.4: HDL Coder Wizard

```
function makeHDL(model2gen, path)
    checkhdl(model2gen)
    makehdl(model2gen,'TargetDirectory',path,
    'SimulationLibPath', path, 'SimulationTool',
    'Cadence Incisive', 'GenerateCoSimModel','Incisive',
    'GenerateValidationModel','off');
end
```

Code 4.1: Make HDL script

The HDL Coder comes with many configurations in different categories:

- Target: Specify if the code should be generated for a specific FPGA or generic ASIC. The target's frequency is also set and when combined with a timing library to define the time taken to do a multiplication for example, timing estimates can be calculated and timing optimizations become possible.
- Optimization settings: Define how the pipelining should happen and the limits of the resource sharing.
- General coding settings and how the clock and reset should function: Coding style and coding standards regarding the clock and reset can be selected to follow predefined industry standard for naming conventions and language versions or completely customized.
- Reporting: The generation can generate a report as Figure 4.5 where the specific components are linked to the code, showing what models made what code. If and how timing, area and optimization reports should be presented.



Contents
Summary
Clock Summary
Code Interface Report
Timing And Area Report
High-level Resource Report
Critical Path Estimation
Optimization Report
Distributed Pipelining
Streaming and Sharing
Delay Balancing
Adaptive Pipelining
Hierarchy Flattening
Target Code Generation
Traceability Report

Generated Source Files
Subsystem.vhd

Referenced Models

HDL Code Generation Report Summary for counter64

Summary

Model	counter64
Model version	1.3
HDL Coder version	3.19
HDL code generated on	2022-05-19 08:25:21
HDL code generated for	Subsystem
Target Language	VHDL
Target Directory	E:\Downloads\2021b-10mayFINAL\2021b-10mayFINAL\tests\counter2

Non-default model properties

CriticalPathEstimation	on
GenerateCoSimModel	Incisive
HDLGenerateWebview	on
HDLSubsystem	counter64/Subsystem
OptimizationReport	on
ResourceReport	on
SimulationTool	Cadence Incisive
TargetDirectory	E:\Downloads\2021b-10mayFINAL\2021b-10mayFINAL\tests\counter2
Traceability	on

Figure 4.5: HDL Coder output reports

- Testbench generation. In the case of the edge detection and DIG-IF systems no HDL testbenches are used since Xcelium connects to Simulink which is used as the testbench. Another way is to generate HDL testbenches which can be tested together with the genereated code for the models in a simulator tool as Xcelium.

4.2.2 Blocksets

In addition to the fundamental blocks provided by the HDL Coder toolbox, additional toolboxes are provided by Mathworks which provides different systems, models and blocks. Each toolbox is specific to certain area or subject where complex systems are created by the interaction of multiple toolboxes. An example of a block provided by Mathworks' DSP HDL Toolbox [13] are generic filters which will be explored more in the following chapters. The filter block is generated from the filter designer [17] and can be configurations to work as any type of filter.

When a design grows and start to contain many different operations and signal lines it can be hard to follow. To abstract out some complexity, subsystems can be made in Simulink. These subsystems are what makes Simulink a hierarchical designer where each subsystem is similar to a entity in VHDL. Subsystems are also be saved and then referenced like having a library of custom blocks, just as toolboxes are used. Subsystem can even be looped meaning one subsystem will be used for every input. Subsystems can also be variant, meaning different subsystem can be used in the same place, assuming they have the same input and output ports. The selection of which subsystem is determined by constants from the workspace or input signals.

An example subsystem in Simulink is a register. The register subsystem consists of a delay used together with a switch as shown in Figure 4.6. Both components supports all types of signals, including arrays of signals, making it a variable size register from the input signal type.

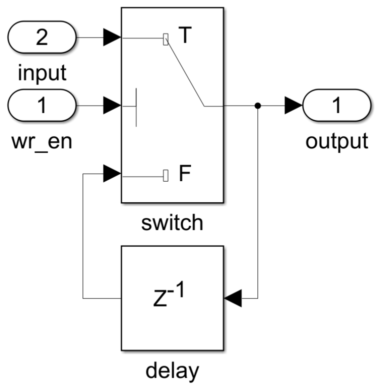


Figure 4.6: Simulink register

An example of a variant subsystem could be a one clock cycle binary index search. An algorithm for finding an available spot in a memory. To create this algorithm, a register keeps track of which spots are available by having a zero at the available indexes and a one at the occupied. Then it is just to find the first zero in the register, which is done by checking if there is a zero in the top or bottom half of the input to each subsystem where the input to the first subsystem is the register content. The design of a 8 bit example is displayed in Figure 4.7.

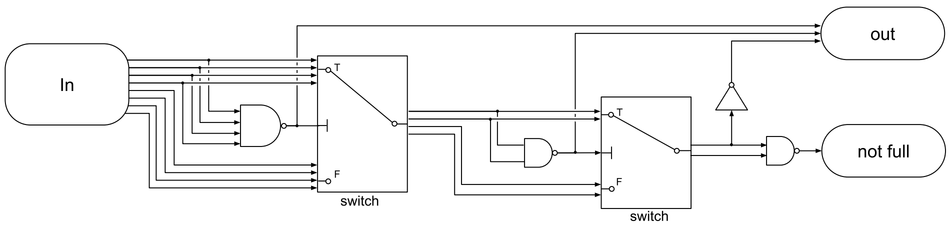


Figure 4.7: 8 bit binary index search

The variant part is that every stage is its own subsystem which are concatenated to create the structure Figure 4.7 to create algorithms handling larger inputs. In this specific case a looped subsystem can not be used since every stage contains a different amount of signals.

4.2.3 Dedicated Blocks

4.2.3.1 HDL Coder Generic RAM

The HDL Coder has dedicated blocks for generic RAMs. The address and data width of the RAMs are defined with the size of the inputs in the Simulink model automatically, which will be generated using generics in HDL.

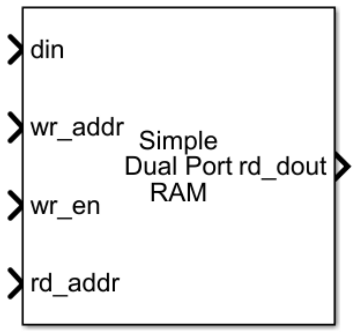


Figure 4.8: HDL Coder Dual port RAM in Simulink

A full description of the generated HDL code for a RAM block can be seen in the appendix C.1, where the line 47 sets the address and data width. The lines from 21 to line 35 calls the components.

4.2.3.2 Block Matrix Multiply

In the DIG-IF vector-matrix multiplication is used to combine or split the signals to or from the antennas. Simulink's matrix multiply block gained fixed point support from the HDL toolbox, letting it perform one clock cycle multiplications by having many multiplication cores. For example combining 8 antennas to 4 output streams requires 32 multiplication cores. If the hardware is not quick enough or the cores take up too much space, another streaming block[16] can be used. By specifying matrix sizes and the hardware its gonna be implemented on, Mathworks supplies the latency and the resources needed.

4.2.3.3 Filter

The easiest way to create a filter is by using Mathworks' filter designer tool. In the tool the filter can be defined either by specifying which implementation along with frequency and magnitude specifications. Mathworks can find the minimum order needed to fulfil the specifications and generate the coefficients. With the addition on HDL Coder, quantization of the filter is possible by defining the arithmetic and fixed point properties of the signal in the filter. When quantifying the filter properties may change due to loss of accuracy in the coefficients leading to the specifications of the filter may need to be retweaked to fit the accuracy in the coefficients.

In the case of the DIG-IF, the properties of each signal are set by the fixed point tool. This increases the control, allowing the signals in each stage of the filter to have different properties. Since a maximum error is set when generating the properties using the fixed point tool, it ensures the filter properties are not changed due to accuracy loss.

4.3 Handwritten HDL Development - Edge Detector

To write the code from a description, the system must be properly understood. Designing the architecture is done by exploring the system with blocks, without a Matlab environment and only a HDL simulator. Time is spent on designing test-benches for specific blocks to ensure its functionality. Different types of statements is used for the handwritten version such as concurrent and sequential signals. In the edge detection, A finite state machine controls the data input and how the data flows in the different registers and operators, having boolean flags raised as indicators.

Using an external SRAM (static random access memory) IP (Intellectual property) will improve the system in terms of area speed and power, compared to a generic RAM made with HDL. But the implementation involves the knowledge of the IPs delays and requires the adjustment of the delays of other components to make sure the output is correct. The delay adjustments may add latency to the overall design. To counteract this in the example of a RAM, a larger read can be done. Different parts of the same output are used for multiple signals. As a result the circuit has become more complex.

Handwritten designs are often hierarchical as well where the design can be split into different files. In the case of the Edge Detector which is not a large model, it was not necessary. The design required description of multiple internal signals, to control the datapaths and registers, and route signals from SRAM IP. More than 60 signals of different widths were used and traced in the handwritten edge detector. The documentation of the code is another task that is not automated and has to be done.

The VHDL code 4.4 shows the routing of one register and the control signals which triggers different datapaths. In line 5, a signal determines if the input of the register is either `RAM_A` or `RAM_B`. Total control of the dataflow is achieved by handwriting the code.

```
---datapaths , routing of Line 1 register
L1_next <= (others => '0') when state_reg = '0'  else L1_R1;
L1_R1 <= (others => '0') when counterLength_reg <= 1  else L1_R2;
L1_R2 <= L1_reg when addressRAM_reg = 0 else L1_R3;
L1_R3 <= unsigned(RAM_A_OUT) when RAMselect_reg ='0' else unsigned(RAM_B_OUT);
```

Code 4.4: Routing of register and control signal

The kernel operator is implemented with simpler functions such as addition and subtraction from different registers which stores parts of each line of the image, as seen in code 4.5. 3 extra identical blocks running in parallel perform the same operation with other inputs, to process 4 pixels in each clock cycle.

```
--matrix convolution, simpler function addition/subtraction
sum1 <= ("0000" & L1_reg(15 downto 8)) + ("0000" & L1_reg(7 downto 0)) +
    ("0000" & L2_reg(7 downto 0)) - ("0000" & L2_reg(23 downto 16)) -
    ("0000" & L3_reg(23 downto 16))- ("0000" & L3_reg(15 downto 8));
sum1_2 <= sum1(10 downto 0) & "0";-- bitshift 2 multiply
```

Code 4.5: Simpler kernel function operation

Verify the code without a Matlab environment, only using the waveforms and output is not an easy task. In the case of the edge detector a testbench is created in VHDL which inserts binary data from a test image. The output is rebuilt to an image and can be viewed for verification. But it is hard to tell which part of the code is wrong when the image is not looking as it should. The only help is when using the HDL simulator. The behavior of each signal is viewed in a timing diagram with its absolute value in a defined radix such as hex as can be seen in Figure 4.9.

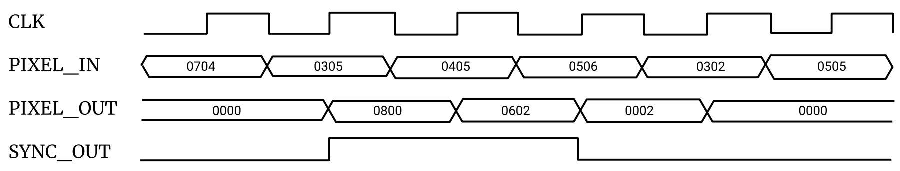


Figure 4.9: Timing diagram from HDL simulator

Chapter 5

Simulation and Synthesis

5.1 HDL Verifier

The HDL Verifier toolbox [18] allows co-simulations of the designs using external simulators such as Cadance's Xcelium. A connection is made from Matlabs or Simulinks environment which enables the co-simulation blocks provided by the HDL Verifier to take an input as a Simulink model and send it to the HDL simulator server where a design is loaded. The server returns the output data to the Simulink models' outputs to be used in the Matlab or Simulink environment as comparison or usage of an external system.

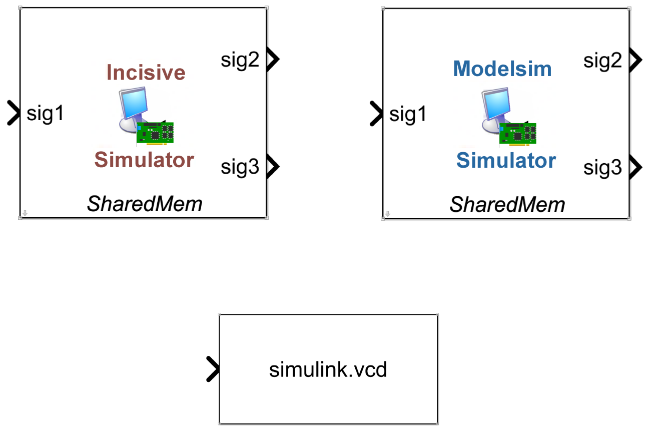
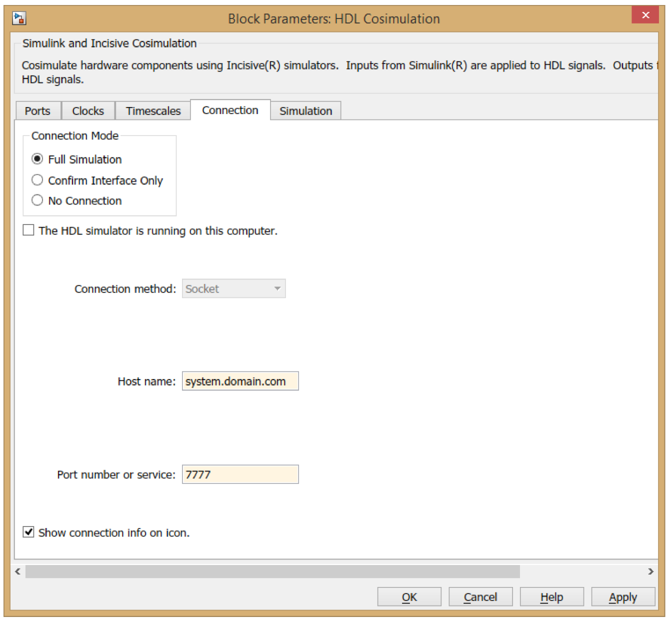


Figure 5.1: Blocks in the HDL Verifier blockset

Models as seen in Figure 5.1 links the Mathworks and Cadence environments creating a powerful simulation environment due to the accessibility of Matlab workspace variables and strength of the simulators processing. Many Simulink models linking to different external simulators can be joined together to create specific testbenches and import other, already compiled designs. These models enable both Matlab researchers and developers and system RTL designers to share the same framework.

The link is established by setting a port or socket in the co-simulation block which allows to run cross-network simulation where the Simulink environment and HDL simulator are executed in different systems. Multiple links are allowed in this format, as long as a port is not shared between different HDL Xcelium instances. The possibility to execute several simulations in parallel in the Simulink environment enables large systems to be integrated and tested at once. Figure 5.2 shows an example setup for a Xcelium remote server at LTH.



Simulink and Incisive Cosimulation

Cosimulate hardware components using Incisive(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs
HDL signals.

Connection Mode

● Full Simulation
○ Confirm Interface Only
○ No Connection

☐ The HDL simulator is running on this computer.

Connection method: Socket

Host name: system.domain.com

Port number or service: 7777

☑ Show connection info on icon.

Figure 5.2: Connection Setup in co-simulation block

5.2 Cadence's Xcelium

Cadence's Xcelium is a simulator for digital hardware in order to verify functionality of RTL or Gate-level designs. This is just one part of the whole process of verification although crucial since after synthesis or layout, the description or architecture can be modified to meet certain constraints and the simulation is the first verification of the RTL logic works as intended.

Figure 5.3 shows how the VHDL or Verilog files are compiled by Xcelium, using commands such as `xmvhdl` or `xmvlog`. Then the command `xmelab` defines the design hierarchy of the model, this command elaborates code for simulation. Lastly, the command `xmsim` simulate the stimuli creating results that can be displayed with another environment such as SimVision with a GUI as displayed in Figure 5.4 or passed back to Mathworks.

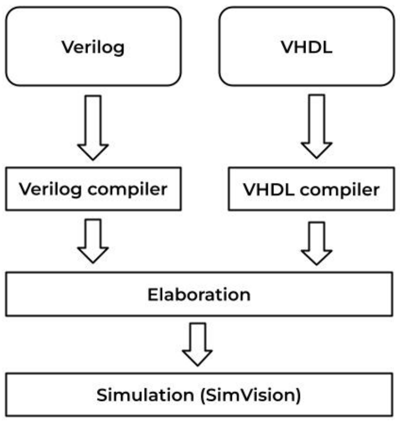


Figure 5.3: Cadence Xcelium environment

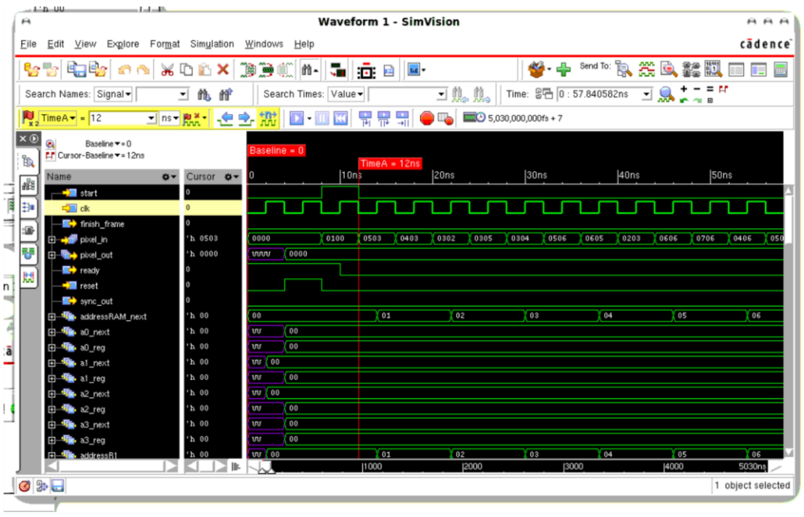


Figure 5.4: Simvision GUI

Cadence's Xcelium is able to run TCL scripts, which enables automation of steps in the simulation. A start up TCL script is used to fetch the license to enable the start of the program. The path to the HDL source files to compile and elaborate and optional configurations are appended as parameters. The configurations can contain if the GUI should be used and on which port the Xcelium should expect the connection to Mathworks. TCL is widely adopted by electronic design automation(EDA) tools makers for automation [11].

5.3 Xcelium Co-simulation

Once Xcelium has the RTL source it can start the simulation. There are several ways to start the HDL simulator and connect to the Mathworks software. Xcelium can be set up as a server or as a client, with Simulink or Matlab respectively. Since Simulink is used as the main environment for system design, Xcelium works as the host waiting for stimuli from Simulink, the client. Figure 5.5 shows the link between the Simulink environment and Xcelium Simulator.

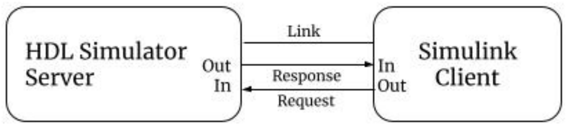


Figure 5.5: Xcelium as a server

With this setup, co-simulation is a tool which enables the sending of data from the Simulink environment to the Xcelium simulator. Since Simulink and Xcelium does not necessarily work at the same rate it is important to set up the timings correctly between the two. Simulink only works in ideal clock cycles while Xcelium simulates the actual time it takes for a signal to change. Having a higher accuracy in Xcelium and not compensating for it in the timing configurations results in the simulation not behaving as intended. Xcelium could be using the same input multiple times and outputs data to Simulink in the the wrong rate.

The co-simulation model generated from HDL Coder in Figure 5.6 shows a comparison between a Mathworks model and a simulated RTL model, where the outputs can be tested with the Simulink tools.

A Matlab automation script handles the generation of co-simulation model, sends the file to the server and then starts the server and simulates the design from using its testbench. The script can be found in appendix B.5. It takes at least two parameters, which model to cosimulate and the address where the server lives. It also has an optional configuration struct which defines the configuration parameters for the server and how to find the testbenches and source files of the model, relative to its path. When the design is done, the script erases the used RTL description on the server and closes the connection. This is done by setting post-simulation TCL commands on the co-simulation block.

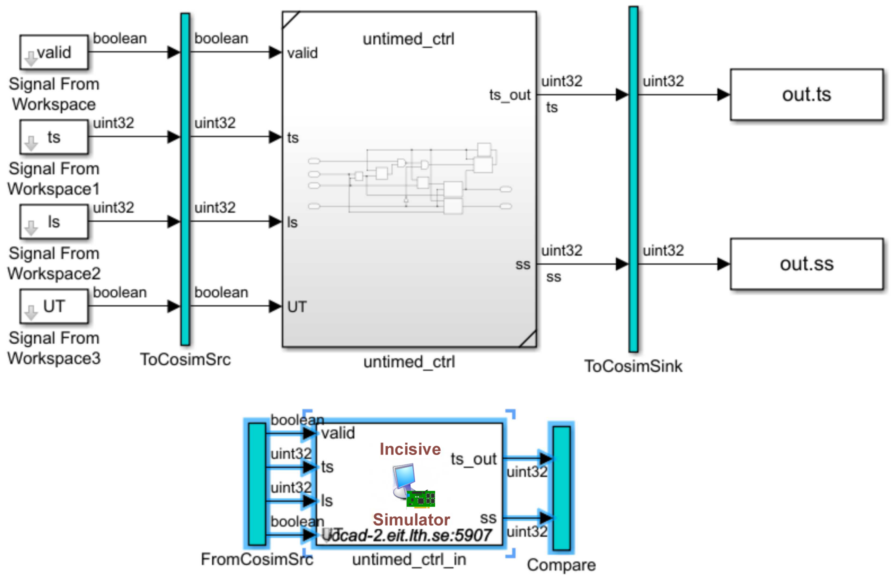


Figure 5.6: Design model and co-simulation linking model

The enabling of Xcelium simulations on a different server in another network is done by first connecting to its Virtual Private Network (VPN). This enable the use of a socket connection from Simulink for transmitting stimuli and receiving output data. The server environment is prepared by uploading the source files which is made by an automatic setup script in Code 5.1 over SSH (Secure Socket Shell). Xcelium is started as host using the server start up script in Code 5.2 ran by the setup script.

```
#!/bin/bash
# Args: name server_files hdl_files server_address server_path port
if (( $# == 6 )); then
  ssh "$4" "mkdir" "$5"
  scp $2 $4:$5
  ssh "$4" "mkdir" "$5$1"
  scp $3 $4:$5$1
  ssh "$4" "cd $5; set NAME=$1; set PORT='$6'; source run_xcelium.tcl;
    /bin/rm -r $1"
else
  echo "Expects 6 arguments:
    name server_files hdl_files server_address server_path port"
  echo "Ex:
    DIG_IF hdl_prj/cosimulation/server_files/* hdl_prj/hdlsrc/DIG_IF_test/*.vhd
    user@server ~/Documents/temp 1337"
  /bin/bash -i
fi
```

Code 5.1: server_setup.sh

```
source setup_xcelium_w_licence
xmvhdl -v93 -smartlib -messages $NAME/*.vhd
xmelab -64bit -access +rwc -vhdl_time_precision 1ns $NAME -messages -mccodegen
xmsim $NAME -64bit -tcl -loadvpi /export/space/local-sw/matlab/instr2019b
/toolbox/edalink/extensions/incisive/linux64/
liblfihdls_gcc41.so:simlinkserver +socket=$PORT
```

Code 5.2: start_xcelium.tcl

The Matlab script in appendix, section B.5 is used to create and setup the co-simulation model also runs the server setup script by using the built in `system` method in Matlab [14] making the entire simulation automated from a single Matlab function.

5.4 Synthesis using Genus

Genus is another Cadence tool, which synthesize a design by elaborate its HDL description, defining the cells used in the description. The synthesis process requires libraries and constraints in order to build a specific architecture from the cells in the library. In the following examples a 65 nm library was used. A digital schematic or netlist is the output from the synthesis which describe how a group of cells is connected by wires. Figure 5.7 displays a visualization of the netlist.

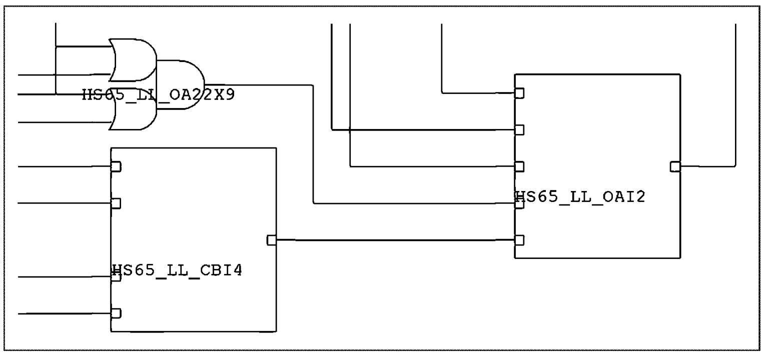


Figure 5.7: Schematic view from Genus

Just as Xcelium, Genus also supports TCL scripts and the Genus synthesis scripts can be found in appendix D. The script include the design setups such as libraries to use, files, clock constraints, synthesis job or effort, the clock and reset configurations and reports on area and timing.

Chapter 6 Results

An overview of our digital design flow can be seen in Figure 6.1. The Figure contains a more specific flow stating which steps are automated and which needs setup, compared the generic flow presented in Figure 2.2. Just as the generic, our flow starts with the description of the concept or product idea, setting up design constants. To enable prototyping with different values for the constants, a script sets up configurations of the different constants with different values. Another script generates test data from the constants to be used for all testbenches. The scripts used to setup and generate the test data for DIG-IF are in appendix B.3 and B.4.

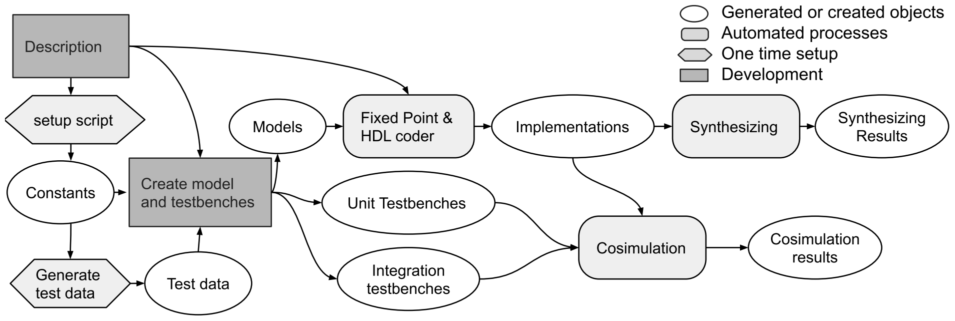


Figure 6.1: Overview of final flow

When a description is made and constants have been set, the development can start. The specific implementation of the edge detector and DIG-IF are described in the following sections, 6.1.1 and 6.1.2 respectively. To have an organised flow, the file structure displayed in Figure 6.2 is used where the different scripts and processes along with the implementations and its testbenches can be found. The full file structure for DIG-IF with all of its components can be found in the Github repository, link in appendix A.2.1.

```
root/
├── hdl_prj/
│   ├── cosimulation/
│   │   ├── cosimulate.m
│   │   ├── server_setup.sh
│   │   └── start_simulator.tcl
│   └── hdl_src/
│       ├── <model name>/
│       │   ├── <generated source file>.vhdl
│       │   └── ...
│       └── ...
├── helper_functions/
├── models/
│   ├── <model name>/
│   │   ├── DUT.slx
│   │   ├── integration_test.slx
│   │   ├── integration_testbench.m
│   │   ├── unit_test.slx
│   │   └── unit_testbench.m
│   └── ...
├── setup/
│   ├── input_creation/
│   │   ├── gen_<input type>.m
│   │   └── ...
│   ├── constants.mat
│   ├── gen_constants.m
│   ├── gen_input_data.m
│   └── input_data.mat
└── test_suite.m
```

Figure 6.2: The flow's file structure

6.1 Models

6.1.1 2D Convolution

In addition to the 2D Convolution Simulink model, a custom Matlab function implementing the same algorithm and Mathworks' `filter2` [15] is also tested. Each using the same testbench. To display the result images in the same format as the input, the output is clamped between 0 and 255 to fit in a 8 bit representation which makes all negative values displayed as black and all values above 255 displayed as white.

The test data used to verify the implementations consists of different images, random noise and an incremental image where the next pixel is brighter than the previous. One example of test data is the image in Figure 6.3 along with the result of the different results for respective implementation.

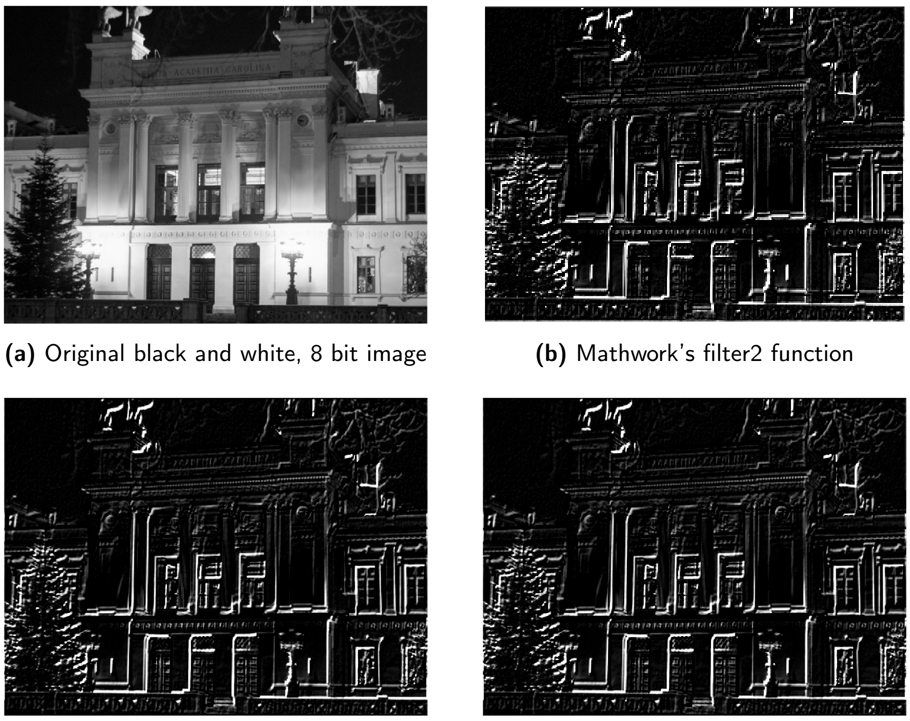

(a) Original black and white, 8 bit image (b) Mathwork's filter2 function (c) Custom Matlab implementation (d) Simulink implementation

Figure 6.3: The 2D convolution result

The keen eye may spot differences in the literal edge cases between the Mathworks' `filter2` version and the other implementations, because there are differences. The implemented algorithm wraps the edges compared to Mathworks' `filter2` which is padded by black edges.

6.1.1.1 HDL Generation

A Matlab implementation of a 2D convolution can look as follows:

```
function g = conv2d_golden(f, w, width)
  [w_height, w_width] = size(w);
  f_amount = size(signal,2)-((w_height-1)*width+w_width-1);
  g = zeros(f_amount, 1);

  for i = 1:f_amount
    for j = 1:w_height
      index = i+(j-1)*width;
      g(i) = g(i) + sum(f(index:index+w_width-1).*w(j,:));
    end
  end
end
```

Code 4.2: 2D convolution golden reference

When running the matlab code through the HDL Coder to generate VHDL code, the result was not feasible since it generated with 2.5 million input pads. This is because the test image shown in Figure 6.4 which is a 8 bit 640 x 480 image was inserted all at once, not streamed a pixel at a time as wanted.

Figure 6.4: Original black and white, 8 bit image

To streamline a function, the variables which needs to be persistent must be explicitly declared. A design was implemented where each function call receives one pixel, having a constant kernel, constant image width and stores the previous pixels in a shift register with reading at certain indexes.

```
function g = gol_mat_stream(x)
    width = 640;
    w = [
        1  1  0;
        1  0 -1;
        0 -1 -1;
    ];

    [w_height, w_width] = size(w);
    persistent memory;
    persistent extraMemory;
    if isempty(memory) || isempty(extraMemory)
        memory = zeros(w_height-1, width);
        extraMemory = zeros(1, w_width-1);
    end

    w_flipped = flip(flip(w,2));
    g = sum([x, extraMemory].*w_flipped(1,:));

    for i=1:w_height-1
        g = g + sum(memory(i, width-w_width+1:width).*w_flipped(i+1,:));
    end

    memory = [
        extraMemory(end), memory(1, 1:end-1);
        memory(1:end-1,end), memory(2:end,1:end-1);
    ];

    extraMemory = [x, extraMemory(1:end-1)];
end
```

Code: 4.3 2D convolution streamed

The design took unusual long time to translate to HDL which led to cancellation of the translation. When the implementation was retried with a smaller image it was revealed that the shifting of the memories are not efficient at all. Instead of a smart shift register, every element of the memory was taken out to an additional register and then inserted back, one step below its previous location.

The 2D convolution implementation continued to Simulink. Instead of custom memories, Simulink's delay blocks were used. Concatenating several delays creates a shift register, as shown the example below in Figure 6.6 and 6.5. A kernel of size 5 by 4 was used in the specific example which can be seen by the numbers of how many signals are used for each input. The image width in the example is 640 pixels wide which of 4 rows are stored in the shift register, show in the outer model, Figure 6.5.

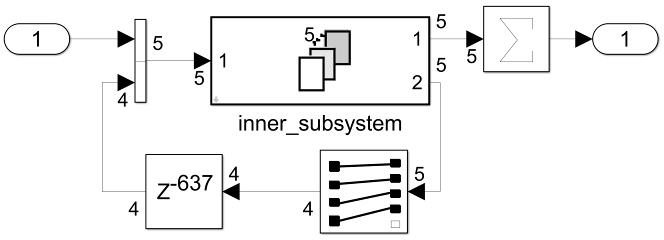


Figure 6.5: Outer model of 2D convolution implemented in Simulink

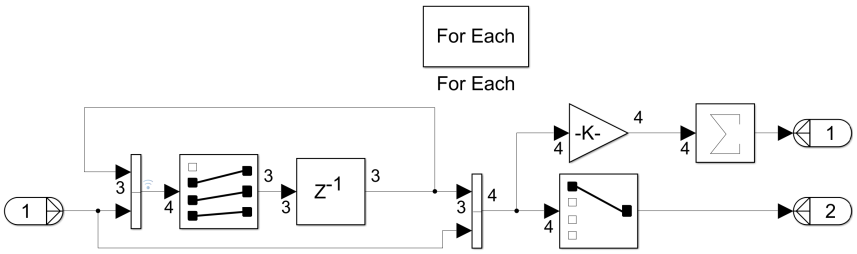


Figure 6.6: Outer subsystem of 2D convolution in Simulink

The Simulink implementation was generated to HDL without any issues.

6.1.1.2 Simulation

The Xcelium simulation of the generated HDL results in the edge cases along the top of the image differs in a second way compared to the results shown in Figure 6.3, as seen in Figure 6.7. This is because the RAMs are not reset between runs. By design the RAMs keeping its values from the last run to increase performance, compared to Simulink's simulations which automatically resets the RAMs between runs. Figure 6.8 displays a zoomed in version of Figure 6.7 on the error which shows that the error disappears after 1280 pixels, corresponding to the first two rows of the 640 pixel wide image that is used.

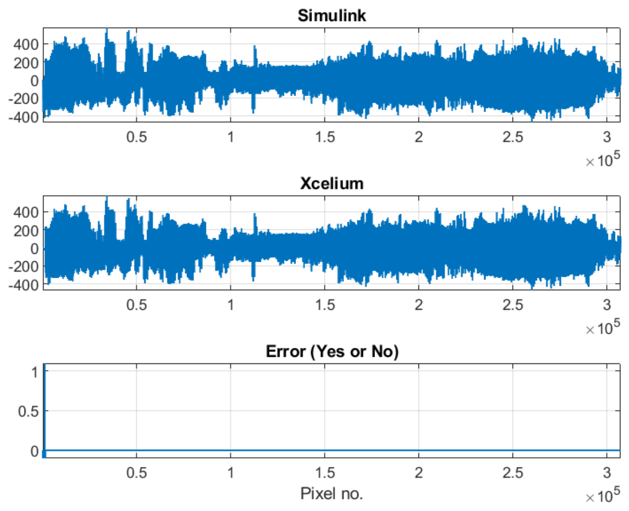


Figure 6.7: Simulink and Xcelium simulation differences

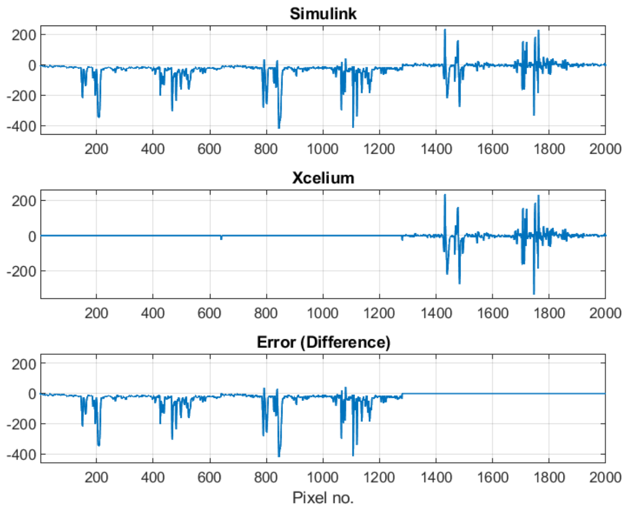
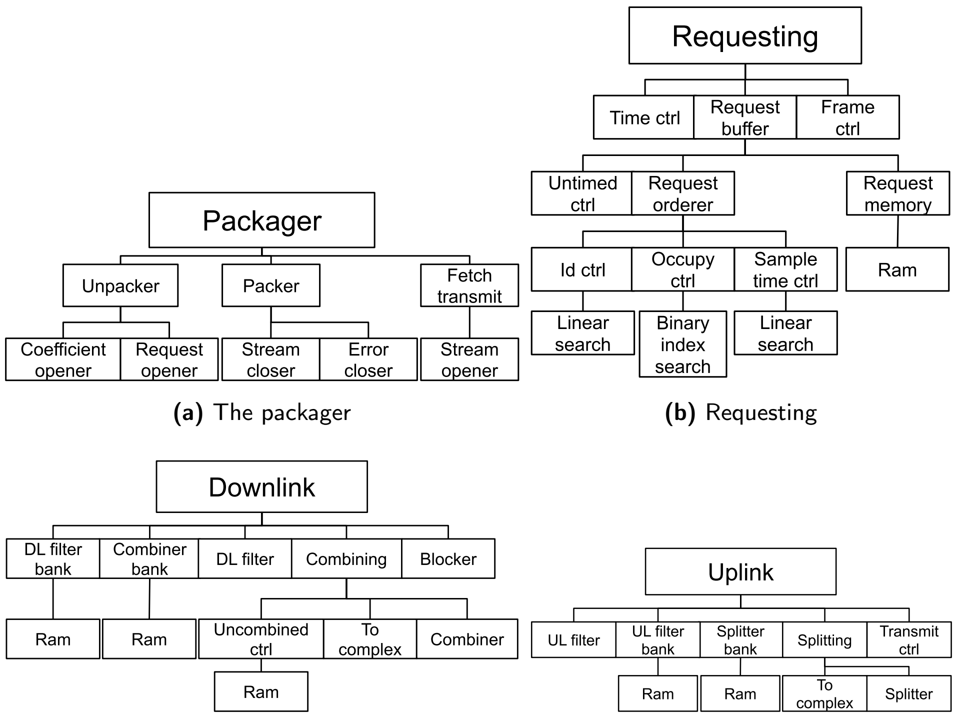


Figure 6.8: Zoomed in Simulink and Xcelium simulation differences

6.1.2 DIG-IF

The DIG-IF is a more complicated system than the edge detection, concatenating many subsystems and models each with its own testbenches. The DIG-IF consists of four main subsystems: The packager, the request control system, the uplink data path and the downlink data path. Each subsystem only contains other subsystem or the fundamental building blocks described in section 4.2.2. The hierarchy in each subsystem is displayed in Figure 6.9. The data width between each block are set in the Simulink environment, and after testbench generation, the HDL Coder engine updates properly all the signals required.



(a) The packager

(b) Requesting

(c) Downlink

(d) Uplink

Figure 6.9: DIG-IF subsystems

6.1.2.1 Requesting

The request control system contains three components: The time control, which count clock cycles as seen in Figure 6.9b. The count can also be shifted with a set time request. A frame controller, which keeps the active transmit or receive request parameters until the clock cycle count is reached and the transmit or receive request is over. And a request buffer. The request buffer inserts all of the transmit and receive requests in a RAM and keeps the request start sample time in an register. The start sample time register is continuously searched to find which request to output next, since the requests may not be inserted in order.

The unit testbench for the request controller verifies that the correct request with all of its parameters are outputted at the correct time. Table 6.1 shows some example incoming requests and Table 6.2 shows a subset corresponding outputs. Each row in the output corresponds to one clock cycle.

ID	fn	UC	SEND	filter addr	dist addr	ts	ls
1	1	F	F	0	0	28	74
2	1	T	F	29	17	103	196
3	1	T	F	29	17	302	194
4	1	F	F	61	3	497	123
5	1	F	F	55	46	621	171
9	1	F	T	6	2	1196	169
7	2	F	T	5	3	880	114
8	2	F	T	5	3	998	188
10	2	F	T	7	1	1366	194
11	2	F	F	32	11	1565	149
12	2	F	F	32	11	1718	56
13	2	F	F	62	37	1782	178
14	2	F	F	21	28	1968	191
⋮							

Table 6.1: Incoming requests

Sample	valid	fn	ID	SEND	filter addr	dist addr
194	T	1	2	F	29	17
195	T	1	2	F	29	17
196	T	1	2	F	29	17
197	F	1	-	-	-	-
198	F	1	-	-	-	-
⋮						
1363	T	2	9	T	5	3
1364	T	2	9	T	5	3
1365	T	2	9	T	5	3
1366	T	2	10	T	7	1
1367	T	2	10	T	7	1
1368	T	2	10	T	7	1

Table 6.2: Part of the request output

Cancelling of request and time shifting is also tested. A cancel request is sent with which ID to cancel and all rows having the specific ID in the expected data are cleared. Table 6.2 shows how the request with an ID of 3 has been canceled. The time shifting is an additional request which adds a specified amount of time. This may corrupt the result by making the response getting shorter or even skip a request due to its start time being in the time that is skipped. All these actions are tested for, to make sure the system does not break. Avoiding corruption of data and skipping request is up to the baseband. In addition specific errors can also be inserted, by inserting requests which will overlap or trying to cancel a request with an ID that does not exists for example. These errors are also tested for and should return an error to the baseband.

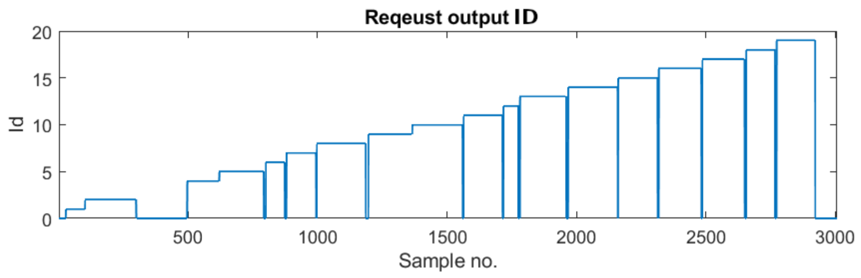


Figure 6.10: Request output ID signal

Figure 6.10 displays the ID of the requests using Xcelium as simulator. The ID is also compared to the expected output ID. ID 3 has also been canceled which is caught both in the expected output and by the simulator and no errors are returned.

The algorithm used to find which request to output next and to find the ID for request canceling, is the same linear search. It works by reading one ID and one start sample time every clock cycle and save its index which also is the request address in the ram. In the case of finding the ID, the occupied flag is lowered when the correct ID is found. If the occupied flag already it continues the search. If no ID to cancel has been found in 64 clock cycles, which is the size of the example memory, an error; ID not found is issued. In the case of finding the request start time, if the time is less than 64 clock cycles away, it is considered found and moved to an output register, waiting to be outputted at the exact right time.

6.1.2.2 Packager

The packager processes the input from the baseband interface to signals inside the circuit. Its hierarchy is seen in Figure 6.9a. The packager also creates packages of responses and errors to send to the baseband. If a type identifier is read, the packager goes into the state which the type identifier specified. It also raises a busy flag, removing the ability to switch state. If the incoming data is a request, specified from the type identifier, the packager collects all of the incoming data in a shift register before outputting it to the request subsystem. The waiting for all of the incoming data introduces a delay, from when a request is inserted to when it can be executed. For most constant configurations, the delay is regarded negligible compared to the request sample sizes.

In all cases the busy flag is raised for a fixed amount of clock cycles defined from the constants, with the exception of when the incoming data is a fetched data stream. Then the busy flag is kept high until the transmit controller in the uplink subsystem defines the transmit is over. The packager also make packages for the baseband. Most packages are just the data with an appended type identifier. The responses packages appends an header as its first output cycle and continue with the data in the response. Just as unpacking incoming data, the busy flag is raised while outputting packages.

6.1.2.3 Up- and Downlink

The up- and downlink as seen in Figure 6.9d and 6.9c keeps its coefficients and weights for the combining, splitting and filtering in different banks. The banks are set by requests on an address from the baseband. The same address is used by the requests to specify which set of coefficients and weights to use for the response. The accuracy for the combiner and splitter is set from the description meaning the fixed point is already defined. This is assuming the previous step in the data path has the correct accuracy. The combiner uses a matrix multiply block. In the case of the splitting this is true since the data comes directly from the baseband. But not necessarily for the combiner since it has a down sampling filter as it previous step. To ensure the filter have enough accuracy while not having to large of an implementation, the fixed point tool is used.

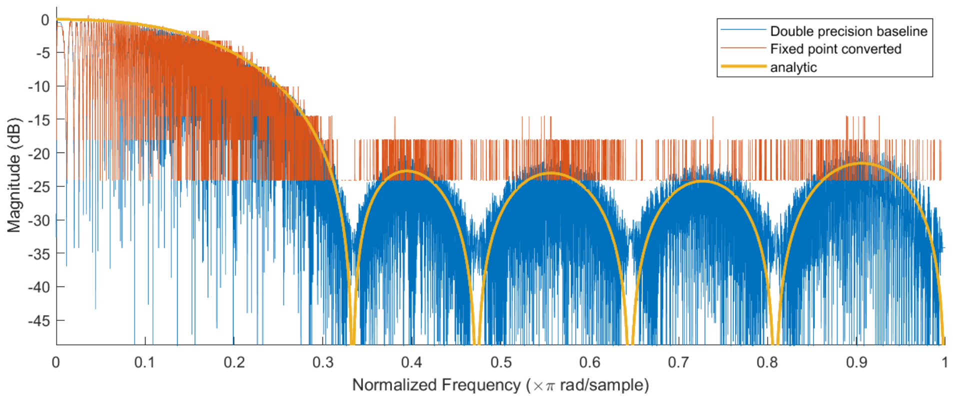


Figure 6.11: fixed point filter results

The filter aims decrease high frequency signals by -20 dB or keep at most 10% of the signals. Having a 6 bit input from the analog to digital converter the fixed point tool is also set to operate with a bitwidth of 6 and only allowed to use binary point scaling and zero bias to ensure the hardware implementation is as small as possible. The incoming signals is represented in the range -1 to 0.96875 which is the range of a signed 6 bit binary number with a fractional length of 5. The fixed point tool rescales the signal after each operation to avoid overflow and as little under flow as possible. Figure 6.11 displays the resulting response and the double precision baseline at a sampling rate of 8000 samples per radian. The input is generated using a chirp meaning the x-axis also shows each sample with the 8000 sample at the 1 mark on the x-axis. This also create the noise below the analytic curve since it is the actual sinusoidal waveform that is displayed. The results show that most of the filtered signal is above -20 dB, at about -18 dB. This is because of $-18 \approx 20 \cdot \log_{10}(2^{-3})$ which means the high frequency signals uses a maximum 3 of the 6 bits and decided enough accuracy in the particular case.

In Figure 6.12 an example input to the combiner is shown. Figure 6.13 displays the combining results when using some example weights shown in the title for each plot. The first weight corresponds to the first antenna, in this case the 1 Hz signal, the second weight for the second, 2 Hz, antenna and so on. In the example a sample rate of 10k sample per second is used with a 6 bit input. The matrix multiplier uses 10 bits for the multiplication and 6 bits for the accumulation resulting in rescaling of the signal compared to the input. From 5 fractional bits to 3.

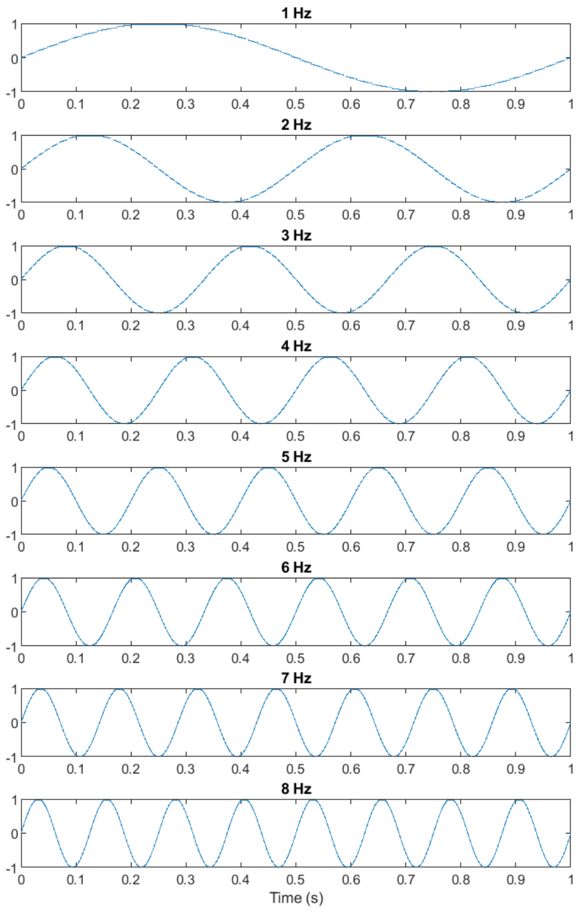


Figure 6.12: combiner input

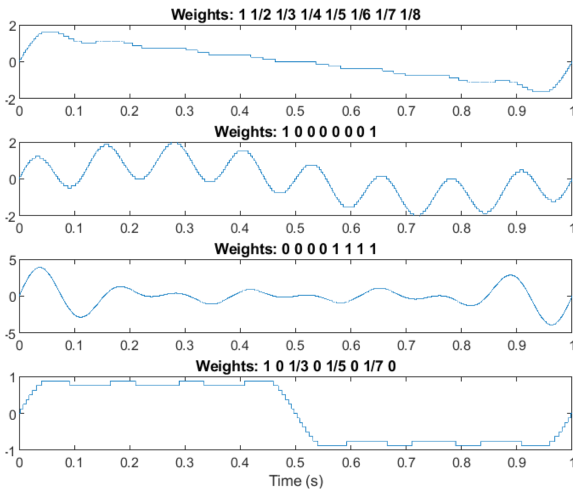


Figure 6.13: combiner results

6.2 Manual translation vs HDL Coder generation

All of the generated code and handwritten code used as examples can be found in the Github repository, link in appendix A.2.1.

6.2.1 HDL Coder Generated

The HDL Coder includes a lot of configuration of how the code will generate which makes changes the appearance and readability of the code. Most of the time the code is readable. The reports with links between each line in the code to the connections and blocks in the model also helps making it clear what each row in the generated code means an do. The only issue is when the model contain a lot of connections that are not in an array or looped, then the generated code will also have many similar, not looped lines making it hard to understand. An example of readable code is the generated code of the RAM models used in DIG-IF. The code can be viewed on the Github repository A.2.1.

6.2.2 Manual Coding

When coding the Edge Detector in VHDL by hand, the architecture is defined by how the data transfers to registers, SRAMs and how operators is connected and configured. Tracing different signals may be problematic due to renaming, syntax errors and logical mistakes from human error. Comments is important in order to improve readability and to show the data flow for readability and understanding for the future. Just as, or even more important is the documentation of the handwritten code which is not automatically generated as for HDL Coder generated created code. As seen in section 4.3 the described function with the registers involves many simpler operations is combined to a rather messy and long syntax.

The verification process can not be automated in the same way as done using the Mathworks tools since the HDL simulator is not connected. Every testbench must be created manually with manually defined input stimuli. The stimuli must then be loaded through the simulator. Using Cadence's Xcelium as simulator would require all the different descriptions, testbenches and input stimuli files to be loaded and compiled. The output from the simulation would be an additional file which most likely needs to be processed before it can be verified to valuate the results. In the case of the Edge detection, another software is needed to rebuild the image from the simulation output. Loading the files into the simulator and preparing the data before is done manually, which could lead to more human error. The different ports and the values of each signal can be analyzed to evaluate the process and results in the timing diagrams, displaying the signals waveform. Since the operation is on each pixel of an image, it is hard to verify the design on a timing diagram which a part of is shown in Figure 6.14.

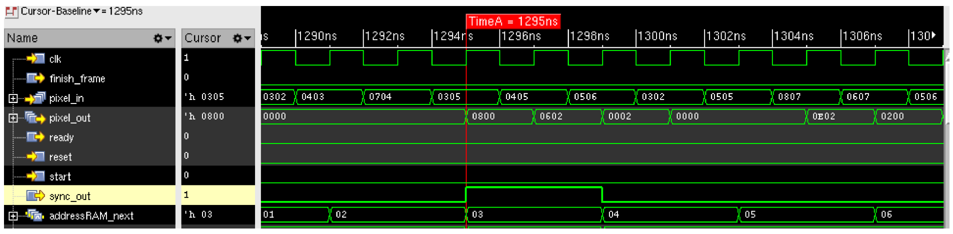


Figure 6.14: Cosimulation waveform of handwritten edge detector

To give the handwritten HDL of the Edge Detection a fair comparison was it also hooked up to Simulink as an external system as displayed in Figure 6.15. With this approach, instead of writing testbenches in VHDL language manually, Simulink is able to generate stimuli for the design under test (DUT) and co-simulate with the handwritten HDL. Since the HDL block behaves as a black box, the output results are sent back to the Simulink environment where it can be analyzed and reconstructed to an image.

In the Simulink model with the Edge Detection as the external system 6.15, a input from the Matlab workspace is created and converted to unsigned integer to be compatible with the HDL co-simulation input. The result from the HDL simulator can be treated as a signal in Simulink and Matlab workspace for future analysis.

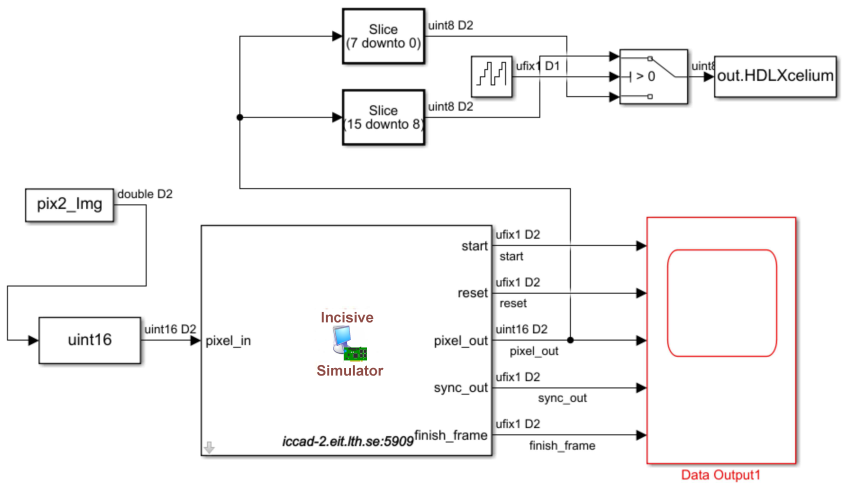


Figure 6.15: Co-simulation on Handwritten HDL

6.3 Genus Synthesis

To verify that the flow is producing quality HDL regarding cost constraints, all models and systems are also synthesized. As mentioned in section 5.4 digital libraries are needed and a 65 nm standard cell library was used to make comparisons between different implementations.

The sections below contains the result of the 2D convolution implementation and several subsystems used for the DIG-IF.

6.3.1 Coefficient Banks

The coefficient banks is general modules which store the weights and coefficients in RAMs that are used with the for the different filter splitter and combination blocks in DIG-IF. The number of instances of each type and area for different types of gates are shown in Table 6.3.

Type	Instances	Area (um^2)	Area (%)
Sequential	205600	2029664.00	77.5
Inverter	38408	60402.68	2.3
Buffer	2116	20248.80	0.8
Logic	94099	508300.52	19.4
Total	340223	2618616	100

Table 6.3: Coefficient Banks RAM gates and area synthesis report

The coefficient banks are implemented using Simulinks generic RAMs which are larger than RAM cores that will be used in a final design.

6.3.2 DIG-IF Requesting

The requesting from DIG_IF was synthesized in order to check the generation of blocks such as time control, frame control and request buffer. The resources used by the synthesis are seen in Table 6.4.

Module	Cell Count	Total Area (um^2)
1.Coeff bank	24433	146080
1.1 Frame control	407	2074
1.2 Request buffer	23865	143132
1.2.1 RAMs	8002	**58370**
1.2.2 Request orderer	15300	**82298**
1.2.3 Untimed control	462	**1909**
1.3 Time control	159	869

Table 6.4: Resources used by coefficient bank control module

6.3.3 Combining Block

The combining block contained in the down link data path does matrix multiplication which is a heavy operation. Table 6.5 displays the size and amount of each component used in the combining block.

Type	Instances	Area (um^2)	Area (%)
Inverter	3550	6780	2.5
buffer	189	751	0.3
logic	45868	260542	97.2
Total	49607	268074	100

Table 6.5: Resource used by combining block

6.3.4 Handwritten Edge Detector VS Generated Edge Detector

Table 6.6 shows the handwritten edge detector and generated edge detector results, both designs include a third party IP sRAM block which was used for synthesis. A 65 nm standard cell library was used.

The generated edge detector uses less resources than the handwritten, a great difference can be seen in the logic area used by generated edge detector which is only 6.57 % to the area used by the handwritten,

Overall in this design, the sRAMs occupy the majority of the circuit, but the generated edge detector reduces its area to 84.06 % compared to the handwritten design

Type	Handwritten	edge detector	Generated	edge detector	Difference
	Instances	Area (um^2)	Instances	Area (um^2)	(Area %)
SRAM IP	2	24993	2	24993	0
Sequential	227	2367	34	409	82.73
Inverter	138	215	14	21	90.24
Buffer	2	4	0	0	-
Logic	586	2891	46	190	93.43
Total	955	30471	96	25616	15.93

Table 6.6: Results from handwritten and generated Edge Detector

Chapter 7

Conclusions

As new improvements have come to smartphones, computers, IoT and other devices, both in processing and data usage, larger and more complex systems have to be made. To keep up with the development, new tools to handle the complexity has also arrived. The main task of this master thesis is to create a design flow using Mathworks environment and Cadence's tools for developing state of the art digital ICs.

7.1 Comparison

The generated description is only as good as the design or architecture. The HDL Coder is not some magic tool, producing perfect code. It just translate the design and helps the development by integrating all of the available tools from Mathworks. To make a good design or architecture, it needs to be more specific compared to an ordinary Simulink model. Mathworks has tools to help define these specifics. Examples are the fixed point tool and the toolboxes containing hardware ready models such as the DSP HDL toolbox.

7.1.1 Quality

The code generated by HDL Coder is good, especially since there are so much configuration on structure and code semantics with industry standards. The documentation is really good since we did not have to write it ourselves and it comes with predefined links between code and model. The only thing it is missing is the ability to comment on why the individual blocks exists and how they function in a higher perspective. But since its easy to have an overview in Simulink which the code links to, those comments may be unnecessary.

The HDL Verifier toolbox has a lot of potential, enabling integration of other HDL simulators and even FPGAs. Existing, handwritten and legacy code made can be integrated and added to the design for co-simulation or as an external model. It also enables an easy way of represent the data output in a proper way like imaging or audio using Mathworks other toolboxes.

Using Xcelium standalone to simulate the edge detectíon took about 40 minutes. This is because the design required a large amount of clock cycles to input all the pixels from a 640x480 image in a serial stream. Using cosimulation from the Simulink environment tested at the simulation time was close to the standalone time.

7.1.2 Developer Experience

Using Simulink is a good way of creating hierarchical design. All models created can be reused such as RAMs or logic circuits, making you create more of a library of components than a single design. Testing and verification can then be done for each model, simplifying the process.

The creation of some systems can sometimes be tedious, renaming or redefining ports of a low level subsystem used in many systems can lead to having to reconnect and edit all of the higher level systems. Another thing is that in Mathworks environment there exists a lot more data types than in HDL. These data types can be nice when debugging but also leads to having to create type conversion models needed only in the Mathworks environment. These conversion models are completely removed in the HDL code generation since on RTL both data types are realized as same data type from the beginning.

7.2 Setup

Setting up the co-simulation server with Xcelium took more time and effort than expected. Working with different networks with multiple security solutions made us have to contact the IT supports multiple times. Mathworks comes with documentation for Incisive, an older version of Xcelium simulator, yet some critical instructions were missing which difficult the task. Even though many steps are similar, multiple keyword commands were updated meaning the old Incisive commands did not work.

For a company, in order to implement a connection between the different programs as in this design flow, the typical setup is a Mathworks client on the developers system and an external Xcelium server. The server would always have the license, not as in our setup where we got a student license which has to be enabled for every simulation. The server setup would also be more straight forward with a simpler method for the server to access the source files.

At the same time the Matlab-Xcelium distributions should be compatible according to Mathworks documentation.

7.3 Future Work

Mathworks have plenty of tools and documentation many of which could improve the flow. Deeper optimization and automation of the HDL generation is always positive. Even more integration of the Cadence tools can also be extended enabling quicker developing times.

7.3.1 Improving the Flow

The HDL Coder does have some optimization configurations such as pipelining and mapping registers to RAM. These were tried but never successfully implemented. The reason is that since the flow is for generic ASIC we had no time and size constraints. When developing for FPGAs, the specific FPGA can be chosen which includes a library of available resources and the timing constraints for the specified targeted frequency. Custom libraries could also be added to enable the optimization but for that, a custom hardware library will be needed. The HDL Coder can also provide timing analyzes such finding as critical path when a custom library is available.

7.3.2 Extending the Flow

Since the HDL Verifier enables integration of other programs and designs an extension of the flow would be to explore other developing environments to produce the actual code. Keeping Mathworks for its environment to set up mathematical golden references and creating testbenches but code VHDL or Verilog in a developing environment made for coding in those languages. Although this would mean that code documentation may not be generated as HDL Coder does automatically.

Many other flows have an additional stage before fabricating an ASIC, testing the design on a FPGA. This could probably be added to the flow rather easily from what is shown in the Mathworks environment. Many vendors of FPGAs have its own toolboxes for its models which provide all data needed for co-simulation. Some toolboxes also states that they can run FPGAs in real time in the Mathworks environment.

An additional stage would also be the routing stage of a design. To automate the tesbenches on routed RTL would give the most accurate cost estimations and would complete the flow for fabrication.

References

[1] Robert Sobot, "Practical Considerations in VLSI IC Design Flow with Respect to Tool Limitations" *IEEE International Symposium on Circuits and Systems (ISCAS)* 2015

[2] O. Z. Batur, M. Koca and G. Dundar, "MATLAB-VHDL design automation for MB-OFDM UWB," *2009 IEEE International Conference on Ultra-Wideband, 2009, pp. 423-427*

[3] K. Ng, "Challenges in using system-level models for RTL verification," *2008 45th ACM/IEEE Design Automation Conference, 2008, pp. 812-815.*

[4] A. Goel, B. B. T. Sundari and S. Mathew, "UVM based Controller Area Network Verification IP (VIP)," *2020 International Conference on Smart Electronics and Communication (ICOSEC),2020, pp. 645-652, doi: 10.1109/ICOSEC49089.2020.9215398.*

[5] T. M. Bhatt and D. McCain, "Matlab as a development environment for FPGA design," *Proceedings. 42nd Design Automation Conference, 2005., 2005, pp. 607-610, doi: 10.1145/1065579.1065737.*

[6] V. Y. Sarges "Evaluating Simulink HDL Coder as a Framework for Flexible and Modular Hardware Description" . *Massachusetts institute of Technology, (2018)*

[7] Dr. R. H. Klenke "Design Flow and Methodology" `http://www.people.vcu.edu/~rhklenke/tutorials/actel/design_flow.html` *Virginia Commonwealth University (1999)*

[8] IEEE 754 Floating point Arithmetic. `https://en.wikipedia.org/wiki/IEEE_754` *Wikipedia (2022).*

[9] Pong P. Chu "RTL Hardware Design Using VHDL Coding for Efficiency, Portability, and Scalability" *Wiley-IEEE Press (2006)*

[10] Digital ASIC Group "Digital ASIC Design" `https://www.eit.lth.se/fileadmin/eit/courses/etin01/manual_etc/dasic.pdf` *Lund University, 2005*

[11] T. L. Brandon, B. F. Cockburn and D. G. Elliott, "HDL2GDS: a fully automated ASIC digital design flow," *Canadian Conference on Electrical and Computer Engineering, 2005., 2005, pp. 1535-1538, doi: 10.1109/C-CECE.2005.1557272.*

[12] Ramdas "VLSI Design – Front End and Back End" `http://verificationexcellence.in/vlsi-design-front-end-and-back-end/`, *Verification Excellence, (2018).*

[13] Mathworks "DSP system toolbox" `https://www.mathworks.com/products/dsp-system.html`

[14] Mathworks "system" command. `https://se.mathworks.com/help/matlab/ref/system.html`

[15] Mathworks "2-D digital filter" `https://se.mathworks.com/help/matlab/ref/filter2.html`

[16] Mathworks "HDL Code Generation for Streaming Matrix Multiply System Object" `https://se.mathworks.com/help/hdlcoder/ug/hdl-code-generation-streaming-matrix-multiply-system-object.html`

[17] Mathworks "filterDesigner" `https://se.mathworks.com/help/signal/ref/filterdesigner-app.html`

[18] Mathworks "HDL Verifier toolbox" `https://se.mathworks.com/products/hdl-verifier.html`

Appendix A
DIG-IF

Github link to Simulink models

A.1 Design Specifics

A.1.1 Requesting

There are four types of data requests, three of which fetches data from the antennas: A normal request of data: `REQ`. Request of uncombined data: `REQ_UC`, which is used to find the symbols. And an untimed request: `REQ_UT`, which is used when the baseband do not know when to sample the data. This could be when the circuit starts up for example.

The parameters for data requests:

Identification number	`id`
Frame number	`fn`
Downlink filter bank address	`dfb`
Uplink filter bank address	`ufb`
Combiner bank address	`cb`
Splitter bank address	`sb`
Start sample time	`ts`
Sample amount	`ls`
Nth transmit/receive data sample	`Dn`

Table A.1: Request data parameters

Note that the uncombined request does not have any combining address parameter, nor does the untimed request have any start sample time parameter.

The fourth type of request `REQ_SEND` is used for transmitting data. Since the baseband does not have exact time tracking, it sends a request to transmit data at a specific time. The DIG-IF will respond with a `FETCH` response at the specified time which will immediately take data from the baseband and transmit it. Exactly how this would work depends on the baseband. One option would be to halt a processor at the baseband to output the contents of an internal memory.

1. Request data: `REQ(id, fn, dfb, cb, ts, ls)`
   (a) Store request until `ts` equals `ti`.
   (b) Filter and combine samples from `ts` to $ss = ts + ls$ of antenna data using `dfb` and `cb`.
   (c) Creates a `RES(id, fn, nb, D0:Dn)` to BB where $nb = ls$ and `D0:Dn` is the processed samples
2. Request uncombined data: `REQ_UC(id, fn, dfb, ts, ls)`
   (a) Store request until `ts` equals `ti`.
   (b) Filters samples from `ts` to $ss = ts + ls$ of antenna data using `dfb`.
   (c) Creates a `RES(id, fn, nb, D0:Dn)` to BB where $nb = ls \cdot \frac{antenna\_num}{streams}$ and `D0:Dn` is the processed samples
3. Request untimed data: REQ_UT(id, fn, dfb, cb, ls)
   (a) Store request last, i.e. create a ts after every request in the queue.
   (b) Filter and combine `ls` samples of antenna data using `dfb` and `cb`.
   (c) Creates a `RES(id, fn, nb, D0:Dn)` to BB where $nb = ls$ and `D0:Dn` is the processed samples
4. Request transmit: `REQ_SEND(id, fn, ufb, sb, ts, ls)`
   (a) Store request until `ts` equals `ti`.
   (b) `FETCH(id, fn, ls)` data from BB

Listing A.1: List of data requests

In addition of data receiving and transmitting requests, there are also a request for canceling other data requests: `REQ_CANCEL`, shift the DIG-IF time counter: `SET_TI`, setting the weights of the combining: `SET_COMBINER` and splitter: `SET_SPLITTER` and setting the filter coefficients: `SET_DL_FILTER`, `SET_UL_FILTER` respectively. The only parameter in a cancel request request is its identification number, just as the only parameter in a; shift of the DIG-IF time counter request, is how much it should shift `tis`.

The requests for setting the coefficients for the filters and weights of the combiner and splitter have similar parameters, the address and the weights or coefficients. But since the address, weights and coefficients are different, both in amount and in bitwidths, the request are split several similar requests.

1. `SET_TI(tis)`

   Set ti to 0 when ti equals tis.
2. `SET_DL_FILTER(dfb, df0:dfn)`

Identification number	`id`
Time shift	`tis`
Down-link filter bank address	`dfb`
Up-link filter bank address	`ufb`
Combiner bank address	`cb`
Splitter bank address	`sb`

Table A.2: Configuration request parameters.

Set downlink filter bank with address dfb with data df0:dfn.

3. `SET_UL_FILTER(ufb, uf0:ufn)`

   Set downlink filter bank with address ufb with data uf0:ufn.

4. `SET_COMBINER(cb, c0:cn)`

   Set combiner bank with address cb with data c0:cn.

5. `SET_SPLITTER(sb, s0:sn)`

   Set splitter bank with address sb with data s0:sn.

6. `CANCEL_REQ(id)`

   Find request with matching id and remove it.

Listing A.2: List of configuration requests

A.1.2 Responding

When it is time to sample data, specified in the request, the DIG-IF creates a response: `RES`, using the same identification number and frame number stated in the request to send the sampled data back to baseband. But if the specified time already has passed in that frame when the request reaches the DIG-IF an negative acknowledgement: `NACK` is sent. If a request overlaps another request, in time, it fails since only one request can be active at once. Since overlaps are not detected until they occurs, the response is not a negative acknowledgement but a failed request: `FAIL`. To differentiate errors, negative acknowledgements are used for immediate feedback whilst failed responses: `FAIL`, are used when an error occurs later. A cancellation request can also fail if it does not find the id. Since the search algorithm to find the correct id takes several clock cycles a fail is sent instead of an negative acknowledgement.

Negative acknowledgements: `NACK(id, fn, error)`

1. Request data: `REQ`
2. Request uncombined data: `REQ_UC`

Fails: `FAIL(id, fn, error)`

1. Request data: `REQ`

2. Request uncombined data: `REQ_UC`
3. Request untimed data: `REQ_UT`
4. Cancelling request: `CANCEL_REQ`

Listing A.3: Error responses

1. `OVERLAP`
2. `TIMING_OUT_OF_BOUND`
3. `REQ_MEM_FULL`
4. `ID_NOT_FOUND`

Listing A.4: Possible errors

A.2 Constraints

A.2.1 Constants

Description	**Name**	**Default value**
Antenna amount	`antenna_num`	8
Output streams	`stream_num`	4
Sample bitwidth	`sample_bitwidth`	6
bitwidth in combination	`pre_sum_combiner_bitwidth`	10
Output stream bitwidth	`post_sum_combiner_bitwidth`	8
Transmit bitwidth	`transmit_bitwidth`	6
Bank address bitwidth	`bank_bitwidth`	64
Downlink filter attributes	`dl_filter_attribs`	{9, false, false, 4, 4}
Uplink filter attributes	`ul_filter_attribs`	{9, false, true, 4, 4}
Combiner attributes	`combiner_attribs`	{32, true, false, 8, 7}
Splitter attributes	`splitter_attribs`	{8, true, false, 8, 7}

Table A.3: Constants

Where the downlink filter, uplink filter, combiner and splitter attributes are a grouping of the following constants:

Amount of coefficients/weights	`coeff_num`
Is the coefficients/weights complex	`complex`
Is the signed coefficients/weights	`signed`
Word length	`word_length`
Fractional part	`fract_length`

Table A.4: Attributes

Note that these constants are only constants in the finalized circuit. During development these can be anything and the entire design changes depending on these.

Appendix B

Matlab code

B.1 Github repository

Link to all used and referenced source code and models on github https://github.com/THammarback/MasterThesis. Note that DIG-IF is not complete.

B.2 Matlab test suite - Script to test all the components

```
clc
addpath(genpath('setup'));
addpath(genpath('helper_functions'));
addpath(genpath('input_creation'));
addpath(genpath('simulink_blocks'));

%% bit64_binary_index_search_test
% No dependencies
clear

%open('bit64_binary_index_search_test');
bit64_binary_index_search_testground;

if any(errors)
   fprintf(2, "ERROR - bit64_binary_index_search:\n");
   disp(errors');
else
    disp("Pass - bit64_binary_index_search");
end

%% find_id_index_test
% No dependencies
clear

%open('find_id_index_test');
find_id_index_testground;

if any(errors)
   disp("ERROR - find_id_index: see errors variable");
   disp(errors');
else
```

```
    disp("Pass - find_id_index");
end

%% untimed_ctrl_test
% No dependencies
clear

%open('untimed_ctrl_test');
untimed_ctrl_testground;
%% Note; because of random, this could happen even
%%though there is no error
if UT_index ~= find(ts+ls ~= out.ss')

    fprintf(2,"ERROR - untimed_ctrl_test")
else
    disp("Pass - untimed_ctrl_test");
end

%% request_buffer_test
% Test depends on: untimed_ctrl
clear

%open('request_buffer_test');
%open('request_buffer_testground');
request_buffer_testground;

if any(any(errors))
   fprintf(2,"ERROR - request_buffer: see errors variable\n");
else
    disp("Pass - request_buffer");
end

%% coeff_bank_test
% Depends on: coeff_bank_multiple_rams, coeff_bank_single_ram
clear

%open('coeff_bank_simple_test');
%open('coeff_bank_simple_testground');
coeff_bank_simple_testground;
if errors
    fprintf(2,"ERROR - coeff_bank_test\n");
else
    disp("Pass - coeff_bank_test");
end

%% coeff_bank_integration_test
% Depends on: coeff_bank_multiple_rams, coeff_bank_single_ram
% Test depends on: frame_ctrl, request_buffer, time_ctrl, package_opener
clear

%open('coeff_bank_integration_test');
coeff_bank_integration_testground;
if errors
    fprintf(2,"ERROR - coeff_bank_integration_test");
else
    disp("Pass - coeff_bank_integration_test");
end

%% receive_blocker
```

```
% No dependencies
clear

%open('receive_blocker_test');
receive_blocker_testground;

if any(errors)
    fprintf(2, "ERROR - receive_blocker_test:\n")
    disp(errors)
else
    disp("Pass - receive_blocker_test");
end


%% filter_block
% Depends on current selected filter (sections2order3)
clear

filter_block_testground;
sim('filter_block_test');
disp("No errors - filter_block_testground - validate the scopes");

%% combiner
% No dependencies
clear

%open('combiner_test');
combiner_testground;

if any(errors)
    fprintf(2, "ERROR - combiner_test:\n")
    disp(errors)
else
    disp("Pass - combiner_test");
end

%% Combining
% Depends on combiner
clear

%open('combining_test');
combining_testground;

if any(errors)
    fprintf(2, "ERROR - combining_test:\n")
    disp(errors)
else
    disp("Pass - combining_test");
end

%% splitter
% No dependencies
clear

%open('splitter_test');
splitter_testground;

if any(errors)
    fprintf(2, "ERROR - splitter_test:\n")
    disp(errors)
else
    disp("Pass - splitter_test");
end

%% to_complex
% No dependencies
clear

%open('to_complex_test');
to_complex_testground;

if any(errors)
    fprintf(2, "ERROR - to_complex_test:\n")
    disp(errors)
else
    disp("Pass - to_complex_test");
end


%% DIG_IF
% Depends on every block (not deprecated ones)
clear

%open('DIG_IF_test');
DIG_IF_testground;

disp("Skip - DIG_IF - Not yet implemented")
%%
disp(" ")
disp("All Done!")
```

B.3 Matlab setup constants script

```
%% Constants
clear;
sample_frequency = 983.04e6; % 983.04MS/s
symbol_size = 4400; %samples in a symbol
frame_size = 10*8*14; %symbols in a frame
downsample = 2;
% frame time = symbol_size*frame_size/(sample_frequency/downsample)
% = 10ms

id_bitwidth = 16;
fn_bitwidth = 16;
ts_bitwidth = 32;
ls_bitwidth = 32;

antenna_num = 8;
stream_num = 4;
sample_bitwidth = 6;
pre_sum_combiner_bitwidth = 10;
post_sum_combiner_bitwidth = 8;
transmit_bitwidth = 6;

req_mem_size = 64; % cant be change without redesign

%% Bank attributes
```

```
%shared
bank_bitwidth = 8;
coeff_ram_bitwidth = 64;

dl_filter_attribs = struct(...
    'name', 'dl_filter', ...
    'coeff_num', 9,...
 % The currently applied dl filter design has 9 coefficients.
    'complex', false,...
    'signed', true,...
    'word_length', 4,...
    'fract_length', 4);

dl_filter_attribs.data_rows = ceil(
    dl_filter_attribs.coeff_num*
    dl_filter_attribs.coeff_bitwidth*
    (1+dl_filter_attribs.complex)/coeff_ram_bitwidth);

ul_filter_attribs = struct(...
    'name', 'ul_filter', ...
    'coeff_num', 10,...
% The currently applied ul filter design has 10 coefficients.
    'complex', false,...
    'signed', true,...
    'word_length', 4,...
    'fract_length', 4);

ul_filter_attribs.data_rows = ceil(
    ul_filter_attribs.coeff_num*
    ul_filter_attribs.coeff_bitwidth*
    (1+ul_filter_attribs.complex)/coeff_ram_bitwidth);

combiner_attribs = struct(...
    'name', 'combiner', ...
    'coeff_num', antenna_num*stream_num,...
    'complex', true,...
    'signed', false,...
    'word_length', 8,...
    'fract_length', 7);

combiner_attribs.data_rows = ceil(
    combiner_attribs.coeff_num*
    combiner_attribs.coeff_bitwidth*
    (1+combiner_attribs.complex)/coeff_ram_bitwidth);

splitter_attribs = struct(...
    'name', 'splitter', ...
    'coeff_num', antenna_num,...
    'complex', true,...
    'signed', false,...
    'word_length', 8,...
    'fract_length', 7);

splitter_attribs.data_rows = ceil(
    splitter_attribs.coeff_num*
    splitter_attribs.coeff_bitwidth*
    (1+splitter_attribs.complex)/coeff_ram_bitwidth);

%%
```

```
save('setup/constants.mat')
```

B.4 Matlab generate test data

```
load('example_mult.mat',
     'example_dl_filter_banks',
     'example_ul_filter_banks',
     'example_combiner_banks',
     'example_splitter_banks')

load('constants.mat',
     'antenna_num',
     'sample_bitwidth',
     'transmit_bitwidth',
     'dl_filter_attribs',
     'ul_filter_attribs',
     'combiner_attribs',
     'splitter_attribs')

req_num = 10;
send_num = 10;
req_index = 1:req_num;
send_index = req_num+1:req_num+send_num;

dl_filter_addresses = (1:numel(example_dl_filter_banks))';
ul_filter_addresses = (1:numel(example_ul_filter_banks))';
combiner_addresses = (1:numel(example_combiner_banks))';
splitter_addresses = (1:numel(example_splitter_banks))';

req_dl_filter_addresses =
     (mod(0:req_num-1, size(dl_filter_addresses, 1))+1)';

req_ul_filter_addresses =
     (mod(0:send_num-1, size(ul_filter_addresses, 1))+1)';

req_combiner_addresses =
     (mod(0:req_num-1, size(combiner_addresses, 1))+1)';

req_splitter_addresses =
     (mod(0:send_num-1, size(splitter_addresses, 1))+1)';

min_req_ls = 5;
max_req_ls = 20;
min_req_gap = 0;
max_req_gap = 10;

new_ti = randi(20)+30;
req_ids = req_index;
send_ids = send_index;

cancel_ids_not_findable = randsample(
     setdiff(1:(req_num+send_num+5), [req_ids, send_ids]),
     5);

cancel_ids = [
     randsample(req_ids, 1),
     randsample(send_ids, 1),
     cancel_ids_not_findable];
```

```matlab

input_set_coeffs = [
    cell2mat(arrayfun(
        @(address, data) set_coeff(dl_filter_attribs, address, data),
        dl_filter_addresses,
        example_dl_filter_banks, 'UniformOutput', false)').';

    cell2mat(arrayfun(
        @(address, data) set_coeff(ul_filter_attribs, address, data),
        ul_filter_addresses,
        example_ul_filter_banks, 'UniformOutput', false)').';

    cell2mat(arrayfun(
        @(address, data) set_coeff(combiner_attribs, address, data),
        combiner_addresses,
        example_combiner_banks, 'UniformOutput', false)').';

    cell2mat(arrayfun(
        @(address, data) set_coeff(splitter_attribs, address, data),
        splitter_addresses,
        example_splitter_banks, 'UniformOutput', false)').';
];

ls = min_req_ls+randi(max_req_ls-min_req_ls, req_num+send_num, 1);
gaps = min_req_gap+randi(max_req_gap-min_req_gap, req_num+send_num, 1);

latency = 4;
%current design time from request on its input pins to it can be read

[ts, ss] = gen_req_timing(ls, gaps, numel(input_set_coeffs)+latency);

stop_time = max(ss);

%input_set_ti = set_ti(new_ti);

input_reqs = [
    arrayfun(@req, ...                    %req
        req_ids', ...                     %id
        ones(req_num, 1)*2, ...           %fn
        req_dl_filter_addresses, ...      %dl filter address
        req_combiner_addresses, ...       %combiner address
        ts(req_index), ...
        ls(req_index));

    arrayfun(@req_send, ...           %req_send
        send_ids', ...                %id
        ones(send_num, 1)*2, ...      %fn
        req_ul_filter_addresses, ... %ul filter address
        req_splitter_addresses, ...  %splitter address
        ts(send_index), ...
        ls(send_index));
];



input_cancel_id = arrayfun(@req_cancel, cancel_ids);
input_transmit_data = transmit_data(
       randi(2^transmit_bitwidth, stop_time, 1)-1
 + 1i*randi(2^transmit_bitwidth, stop_time, 1)-1i);
```

```matlab

antenna_input_data = fi(
          randi(2^sample_bitwidth, stop_time, antenna_num)-1
    + 1i*randi(2^sample_bitwidth, stop_time, antenna_num)-1i,
false, sample_bitwidth, 0);

input_bus = make_input(input_set_coeffs, input_reqs, input_cancel_id);

%% expected_output
cancel_ids_not_found = arrayfun(
    @(cancel_id) any([req_ids, send_ids] == cancel_id),
    cancel_ids);

expected_valid = assign_booleans(arrayfun(
    @(id) all(id~=cancel_ids(cancel_ids_not_found)),
    [req_ids, send_ids]),
    ts, ss);

expected_id = assign_numbers(
    [req_ids, send_ids].*arrayfun(
        @(id) all(id~=cancel_ids(cancel_ids_not_found)),
        [req_ids, send_ids]),
    ts,ss);

expected_SEND = assign_booleans(
    [false(1, req_num), true(1, send_num)],
    ts, ss);

expected_filter_address = assign_numbers(
    [req_dl_filter_addresses, req_ul_filter_addresses],
    ts, ss);

expected_dist_address = assign_numbers(
    [req_combiner_addresses, req_splitter_addresses],
    ts, ss);

expected_output = struct(
    'req', table(
        expected_valid.',
        expected_id.',
        expected_SEND.',
        expected_filter_address.',
        expected_dist_address.',
        'VariableNames', {
            'valid',
            'id',
            'SEND',
            'filter_address',
            'dist_address'
        }),
    'errors', struct(
        'id_not_found',
        cancel_ids_not_findable));

save('setup/example_input.mat',
    'stop_time',
    'input_bus',
    'antenna_input_data',
    'input_transmit_data',
    'expected_output');
```

B.5 Cosimulation script in matlab

```
function cosimulate(name, server_address, options)
% COSIMULATE a simulink block with a hdl simulator such as Xcelium.
%   1. Generates a cosim model
%   2. Start simulation server
%      @ xcelium_server_go.sh
%      - Copy the run command files to the server
%      - Copy the hdl files to the server
%      - Starts the server
%   3. Run the cosimulation and
%      compare Simulink with co-simulator
%
%   cosimulate('simulink_model', user@server);
%
%   Insert optional options as a struct. Default values:
%   cosimulate('simulink_model', user@server, struct(...
%       'port', '5908',...
%       'server_files', [root, '/server_files/'],...
%       'tb_extension', _test,...
%       'hdl_file_path', ['hdl_prj/hdlsrc/', name, tb_extension, '/'],...
%       'server_path', '~/Documents/xcelium_cosim/',...
%   ));


    windows_root = erase(fileparts(mfilename('fullpath')),[pwd '\']);
    root = windows_root;
    root(strfind(root,'\'))='/';

    if exist('options', 'var') && isfield(options, 'port')
        port = string(options.port);
    else
        port = '5908';
    end

    if exist('options', 'var') && isfield(options, 'server_files')
        server_files = options.server_path;
    else
        server_files =  [root, '/server_files/'];
    end

    if exist('options', 'var') && isfield(options, 'tb_extension')
        tb_extension = options.tb_extension;
    else
        tb_extension = '_test';
    end

    if exist('options', 'var') && isfield(options, 'file_path')
     hdl_file_path = options.file_path;
    else
     hdl_file_path = ['hdl_prj/hdlsrc/', name, tb_extension, '/'];
    end

    if exist('options', 'var') && isfield(options, 'server_path')
        server_path = options.server_path;
    else
```

```
        server_path = '~/Documents/xcelium_cosim/';
    end

    open([name, '_test']);
    makehdltb([name, tb_extension,'/' , name],
    'GenerateCoSimModel','Incisive');

    set_param(['gm_' name tb_extension '_in/' name '_in'],
    'CommLocal', 0);
    set_param(['gm_' name tb_extension '_in/' name '_in'],
    'CommHostName', extractAfter(server_address, "@"));
    set_param(['gm_' name tb_extension '_in/' name '_in'],
    'CommPortNumber', port);
    set_param(['gm_' name tb_extension '_in/' name '_in'],
    'TclPostSimCommand', 'after␣1␣{exit}');

    cmd = ['start␣' windows_root '\xcelium_server_go.sh␣' name
    '␣"' server_files '*"␣"' hdl_file_path '*.vhd"␣'
    server_address '␣' server_path '␣' port];
    system(cmd);
    loop_try_sim(5);

function loop_try_sim(max_loops)
    try
        out = sim(['gm_' name '_test_in']);
    catch error
     if strcmp(error.identifier,'Simulink:SFunctions:SFcnErrorStatus')
         loop_try_sim(max_loops-1);
     else
         rethrow(error)
      end
    end
end
end
```

Appendix C

VHDL code

Github link to vhdl code

C.1 HDL Coder RAM block

```
-- Generated by MATLAB 9.11 and HDL Coder 3.19
-- Module: rams_block
-- Hierarchy Level: 3
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.coeff_bank_pac.ALL;

ENTITY rams_block IS
  PORT( clk      :   IN     std_logic;
        enb      :   IN     std_logic;
        wr_data  :   IN     std_logic_vector(63 DOWNTO 0);
        wr_addr  :   IN     std_logic_vector(7 DOWNTO 0);
        wr_en    :   IN     std_logic;
        rd_addr  :   IN     std_logic_vector(7 DOWNTO 0);
        Out1     :   OUT    vector_of_std_logic_vector8(0 TO 7));
END rams_block;

ARCHITECTURE rtl OF rams_block IS
  -- Component Declarations
COMPONENT SimpleDualPortRAM_generic
GENERIC( AddrWidth              : integer;
         DataWidth              : integer );
PORT( clk                         :   IN     std_logic;
enb                               :   IN     std_logic;
wr_din  : IN std_logic_vector(DataWidth - 1 DOWNTO 0);
wr_addr  : IN std_logic_vector(AddrWidth - 1 DOWNTO 0);
wr_en                             :   IN     std_logic;
rd_addr  : IN std_logic_vector(AddrWidth - 1 DOWNTO 0);
rd_dout  :  OUT    std_logic_vector(DataWidth - 1 DOWNTO 0));
END COMPONENT;
COMPONENT fi_splitter_block4
 PORT( in_word  : IN     std_logic_vector(63 DOWNTO 0);
       out_arr  : OUT    vector_of_std_logic_vector8(0 TO 7));
END COMPONENT;
```

```
  -- Component Configuration Statements
  FOR ALL : SimpleDualPortRAM_generic
    USE ENTITY work.SimpleDualPortRAM_generic(rtl);
  FOR ALL : fi_splitter_block4
    USE ENTITY work.fi_splitter_block4(rtl);
  -- Signals
SIGNAL Simple_Dual_Port_RAM_System_out1 :
 std_logic_vector(63 DOWNTO 0);
SIGNAL out_rsvd   : vector_of_std_logic_vector8(0 TO 7);

BEGIN
  USimple_Dual_Port_RAM_System_2 : SimpleDualPortRAM_generic
    GENERIC MAP( AddrWidth => 8, DataWidth => 64)
    PORT MAP(  clk => clk,
               enb => enb,
               wr_din => wr_data,
               wr_addr => wr_addr,
               wr_en => wr_en,
               rd_addr => rd_addr,
               rd_dout => Simple_Dual_Port_RAM_System_out1);
  Ufi_splitter_4 : fi_splitter_block4
    PORT MAP(  in_word => Simple_Dual_Port_RAM_System_out1,  --
               out_arr => out_rsvd);
  Out1 <= out_rsvd;
END rtl;
```

Appendix D

TCL scripts for synthesis

TCL commands for Genus

D.1 synth.tcl

```
#Define ROOT to the current folder where the project is
#scripts are placed in the folder scripts

#set ROOT "~/Desktop/Your_project_directory"

set SYNT_SCRIPT    "${ROOT}/scripts"
set SYNT_OUT       "${ROOT}/OUTPUTS"
set SYNT_REPORT    "${ROOT}/REPORTS"

puts "\n\n\n DESIGN FILES \n\n\n"
source $SYNT_SCRIPT/design_setup.tcl

puts "\n\n\n ANALYZE HDL DESIGN \n\n\n"
read_hdl -vhdl ${Design_Files}

puts "\n\n\n ELABORATE \n\n\n"
elaborate ${DESIGN}

check_design
report timing -lint

puts "\n\n\n TIMING CONSTRAINTS \n\n\n"
source $SYNT_SCRIPT/create_clock.tcl

puts "\n\n\n SYN_GENERIC \n\n\n"
syn_generic

puts "\n\n\n SYN_MAP \n\n\n"
```

```
syn_map

puts "\n\n\n SYN_OPT \n\n\n"
syn_opt

report_summary -outdir $SYNT_REPORT

puts "\n\n\n EXPORT DESIGN \n\n\n"
write_hdl    > ${SYNT_OUT}/${DESIGN}.v
write_sdc    > ${SYNT_OUT}/${DESIGN}.sdc
write_sdf   -version 2.1  > ${SYNT_OUT}/${DESIGN}.sdf

puts "\n\n\n REPORTING \n\n\n"
report qor      > $SYNT_REPORT/qor_${DESIGN}.rpt
report area     > $SYNT_REPORT/area_${DESIGN}.rpt
report datapath > $SYNT_REPORT/datapath_${DESIGN}.rpt
report messages > $SYNT_REPORT/messages_${DESIGN}.rpt
report gates    > $SYNT_REPORT/gates_${DESIGN}.rpt
report timing   > $SYNT_REPORT/timing_${DESIGN}.rpt
```

D.2 Design_setup.tcl

```
# Set the top_entity_name you want to do synthesis on
set DESIGN $top_entity_name
set DESIGN_FILES_PATH "${ROOT}/hdl_src/"
set_attribute script_search_path ${SYNT_SCRIPT}
set_attribute init_hdl_search_path ${DESIGN_FILES_PATH}
set_attribute init_lib_search_path { \
/usr/local-eit/cad2/cmpstm/stm065v536/CORE65LPLVT_5.1/libs \
/usr/local-eit/cad2/cmpstm/stm065v536/CLOCK65LPLVT_3.1/libs \
} /

set_attribute library { \
CLOCK65LPLVT_wc_1.00V_125C.lib \
CORE65LPLVT_wc_1.00V_125C.lib \
} /

# put all your design files here
set Design_Files " ${DESIGN_FILES_PATH}main_entitiy.vhd "
set SYN_EFF medium
set MAP_EFF medium
set OPT_EFF medium
set_attribute syn_generic_effort ${SYN_EFF}
set_attribute syn_map_effort ${MAP_EFF}
set_attribute syn_opt_effort ${OPT_EFF}
set_attribute information_level 5
```

D.3 create_clock.tcl

```
#values in picoseconds
set PERIOD 4000
set ClkTop ${DESIGN}
set ClkDomain ${DESIGN}
set ClkName clk
set ClkLatency 200
set ClkRise_uncertainty 200
set ClkFall_uncertainty 200
set ClkSlew 200
set InputDelay 200
set OutputDelay 200

#check the name of the entity in the next line
#/designs/ENTITY/ports_in/clk
define_clock -name $ClkName -period $PERIOD -design $ClkTop -domain
 $ClkDomain [find /designs/combiner/ports_in/clk]
set_attribute clock_network_late_latency $ClkLatency $ClkName
set_attribute clock_source_late_latency  $ClkLatency $ClkName
set_attribute clock_setup_uncertainty $ClkLatency $ClkName
set_attribute clock_hold_uncertainty $ClkLatency $ClkName
set_attribute slew_rise $ClkRise_uncertainty $ClkName
set_attribute slew_fall $ClkFall_uncertainty $ClkName
external_delay -input $InputDelay  -clock [find / -clock $ClkName]
-name in_con  [find /des* -port ports_in/*]
external_delay -output $OutputDelay -clock [find / -clock $ClkName]
-name out_con [find /des* -port ports_out/*]
```

Printed by Tryckeriet i E-huset, Lund 2022

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-891
http://www.eit.lth.se