MASTER'S THESIS

RAPID PROTOTYPING FROM ALGORITHM TO FPGA PROTOTYPE

Author	Joonas Järviluoma
Thesis supervisor	Antti Mäntyniemi
Second supervisor	Jukka Lahti
Thesis technical supervisor	Esa-Matti Turtinen

August 2015

ABSTRACT

Wireless data usage continuously increases in today's world setting higher requirements for wireless networks. Ever increasing requirements result in more complex hardware (HW) implementation, especially telecommunication System-on-Chips (SoC) performance is playing a key-role in this development. Complexity increases design workload, therefore, it makes design flow times longer. High-Level Synthesis (HLS) tools have been designed to automate and accelerate design by moving manual work on a higher level.

This Master's Thesis studies MathWorks HLS workflow usage for rapid prototyping of Wireless Communication SoC Intellectual Property (IP). This thesis introduces design and FPGA prototyping flow of Application-Specific Integrated Circuit (ASIC). It presents good design practices targeted for HLS. It also studies MathWorks Hardware Description Language (HDL) generation flow with HDL Coder, possible problems during the flow and solutions to overcome the problems. The HLS flow is examined with an example design that scales and limits the power of IQ-data. This work verifies the design in a Field-Programmable Gate Array (FPGA) environment. It concentrates on evaluating the usage and benefits of MathWorks HLS workflow targeted for rapid prototyping of SoCs.

The Example IP is a Simulink model containing MATLAB algorithms and System Objects. The design is optimized on algorithm level and synthesized into VHDL. The generated Register-Transfer Level (RTL) is verified in co-simulation against the algorithm model. Optimization and verification methods are evaluated. The HDL model is further processed through logic-synthesis using the 3rd party synthesis tool run automatically with a script created by MathWorks workflow. The generated design is tested on FPGA with FPGA-in-the-loop simulation configuration. FPGA prototyping flow benefits for rapid prototyping are evaluated.

Coding styles to generate synthesizable HDL code and simulation methods to improve simulation speed of hardware-like algorithm were discussed. MathWorks HLS workflow was evaluated for rapid prototype purposes from algorithm to FPGA. Optimization methods and capability for production quality RTL for ASIC target were also discussed.

MathWorks' tool flow provided promising results for rapid prototyping. It generated human-readable HDL that was successfully synthesized on FPGA. The FPGA model was simulated in FPGA-in-the-loop configuration successfully. It also provided good area and speed results for the ASIC target when the algorithm was written strictly from the hardware perspective. The process was found to be distinct and efficient.

Keywords: HDL, HLS, FPGA prototyping, algorithm, rapid prototyping, MATLAB, HDL Coder

TIIVISTELMÄ

Langattoman datan käyttö kasvaa jatkuvasti nykymaailmassa ja asettaa korkeammat vaatimukset langattomille verkoille. Kasvavat vaatimukset tekevät laitteistototeutuksesta kompleksisempaa, erityisesti tietoliikenteessä käytettävien järjestelmäpiirien (SoC) tehokkuus on avainasemassa. Tämä kasvattaa suunnittelun työmäärää ja näin ollen suunnitteluvuohon kuluva aika pidentyy. Korkean tason synteesi (HLS) on kehitetty automatisoimaan ja nopeuttamaan digitaalisuunnittelua siirtämällä manuaalista työtä korkeammalle tasolle.

Tämä diplomityö tutkii MathWorks:n HLS-vuon käyttöä langattomaan viestintään suunniteltavien SoC:ien tekijänoikeudenalaisten standardoitujen lohkojen (IP) nopeaan prototypointiin. Työ esittelee perinteisen asiakaspiirin (ASIC) suunnitteluvuon, FPGA-prototypointivuon ja suunnitteluperiaatteet HLS:ää varten. Työssä käydään läpi MathWorks:n laitteistokuvauskielen (HDL) generointivuo HDL Coder:lla, mahdollisia ongelmakohtia vuossa ja ratkaisuja ongelmiin. HLS-vuota tutkitaan esimerkkimallin avulla, joka skaalaa ja rajoittaa IQ-datan tehoa. Esimerkkimallin toiminta tarkistetaan ohjelmoitavan logiikkapiirin (FPGA) kanssa. Työ keskittyy arvioimaan MathWorks:n HLS-vuon käyttöä ja hyötyä nopeaan prototypointiin SoC:ien kehityksessä.

Esimerkkinä käytetään Simulink-mallia, joka sisältää MATLAB-funktioita ja System Object-olioita. Algoritmitasolla optimoitu malli syntesoidaan VHDL:ksi ja rekisterinsiirtotason (RTL) mallin toiminta tarkistetaan yhteissimulaatiolla alkuperäistä algoritmimallia vasten. Optimointi- ja verifiointimenetelmien toimivuutta ja tehokkuutta arvioidaan. Generoitu HDL-malli syntesoidaan kolmannen osapuolen logiikkasynteesi-työkalulla, joka käynnistetään MathWorks:n työkaluvuon generoimalla komentosarjalla. Luotu malli ohjelmoidaan FPGA:lle ja sen toiminta tarkistetaan FPGA-simulaatiolla.

Syntesoituvan HDL-koodin generointiin vaadittavia koodaustyylejä ja algoritmimallin simulointinopeutta parantavia menetelmiä tutkittiin. MathWorks:n HLS-vuon soveltuvuutta nopeaan prototypointiin algoritmista FPGA-prototyypiksi pohdittiin. Lisäksi optimointimenetelmiä ja vuon soveltuvuutta tuotantolaatuisen RTL:n generoimiseen arvioitiin.

MathWorks:n työkaluvuo osoitti lupaavia tuloksia nopean prototypoinnin näkökulmasta. Se loi luettavaa HDL-koodia, joka syntesoitui FPGA:lle. Malli ajettiin onnistuneesti FPGA:lla. Vuon avulla saavutettiin hyviä tuloksia pinta-alan ja nopeuden suhteen, kun malli optimoitiin asiakaspiirille. Tämä vaati mallin kuvaamista tarkasti laitteiston näkökulmasta. Prosessi oli kokonaisuudessaan selkeä ja tehokas.

Avainsanat: laitteistokuvauskieli, korkean tason synteesi, FPGA-prototypointi, algoritmi, nopea prototypointi, MATLAB, HDL Coder

TABLE OF CONTENTS

FOREWORD

LIST OF ABBREVIATIONS AND SYMBOLS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CLB	Configurable Logic Block
DDR3	Double Data Rate 3 (type of memory)
DL	Downlink
DUT	Device Under Test
EDA	Electronic Design Automation
FF	Flip-Flop
FIL	FPGA-in-the-loop
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
IC	Integrated Circuit
I/O	Input and Output
IP	Intellectual Property
JTAG	Joint Test Action Group
LUT	Look-Up Table
MEX	MATLAB Executable
OOP	Object-Oriented Programming
PAR	Place and Route
PCI-E	Peripheral Component Interconnect Express
PVT	Process, Voltage and Temperature
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register-Transfer Level
SATA	Serial Advanced Technology Attachment
SoC	System-on-Chip
SRAM	Static Random Access Memory
STA	Static Timing Analysis
SW	Software
UMRBus	Universal Multi-Resource Bus
2G	Second Generation (mobile network)
3G	Third Generation (mobile network)
4G	Fourth Generation (mobile network)

bps	bits per second
GB	Gigabyte
MB	Megabyte

1.   INTRODUCTION

In 1975, Gordon E. Moore made forecast that the number of transistors that can be placed on an Integrated Circuit (IC) will double every 24 months. This trend has held true already over half a century because of continuous and increasing competition in semiconductor industry. One of the major areas that drive the semiconductor industry evolution is telecommunications, especially mobile broadband systems. [1]

Mobile access to internet was first introduced in Second Generation (2G) of mobile phone technology in 1991. Third Generation (3G) was introduced in 2001 and Fourth Generation (4G) in 2006, and the need for wireless internet access by mobile devices has increased exponentially since on. This evolution has pushed the boundaries of mobile broadband systems, therefore, the requirements for mobile networks solutions have increased in rapid pace. This drives the competition between telecommunication companies harder and digital HW evolution faster. This means that ASICs have to be able to process more data in shorter time. [2]

ASIC designs are getting more complex continuously, not only due to the increasing need of performance and functionality, but also stricter requirements on size and power-efficiency. This leads to increasing workload on design and verification. In today's ASIC development, verification has become the most time consuming part of the design flow. The growth of design sizes and workload increases design flow times and affects productivity. Minimizing these is one of the key-points for profitable SoC business. Figure 1 below presents the trend of workload of manual phases as a function of design complexity. [3][4]
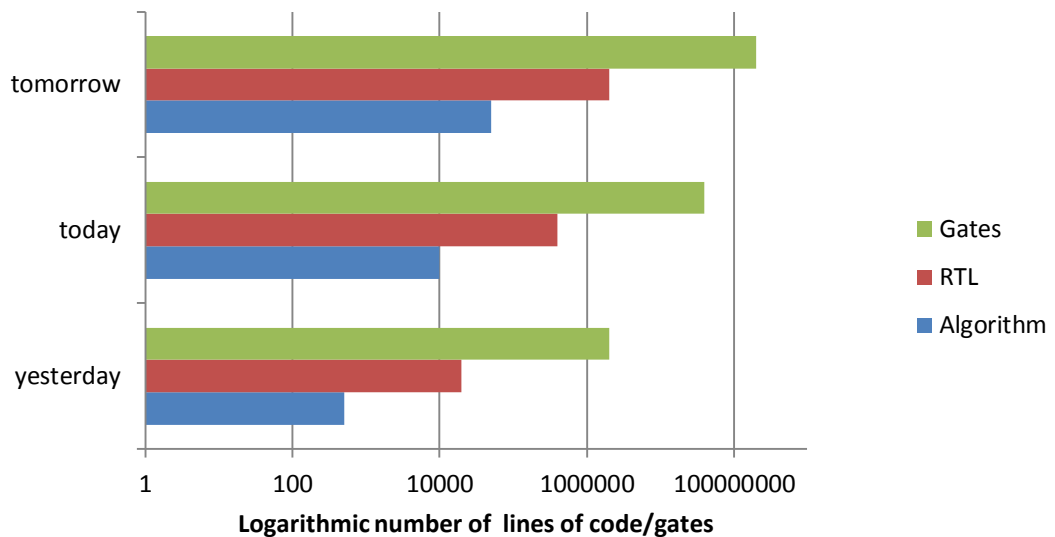
Figure 1. Trend for lines of code as a function of complexity.

ASIC development consists of several phases from system specifications to actual chip. Classically, this includes lot of manual work on coding and verification of the system. HLS tools have been introduced to partly automate the code generation and to ease verification to speed up the design flow by moving the design focus on a higher level. Another improvement has been FPGA prototyping to early test the system

functionality in real-time environment to avoid expensive re-spins in ASIC manufacturing. Complex designs have also improved the IP reuse to avoid spending precious time on designing the same blocks multiple times in different projects. [3][5]

In this work, rapid prototyping of telecommunication SoC IP design is studied with HDL Coder and HDL Verifier tools provided by MathWorks. HDL coder is a HLS tool that can be used to generate HDL code from Simulink and MATLAB algorithms and further process the generated HDL code into a FPGA netlist together with 3rd party synthesis tools. FPGA environment is used for real-time testing of the synthesized design.

This work evaluates the MathWorks HLS flow for rapid prototyping; time and workload benefits it provides, possible problems designer may encounter and solutions to overcome those for faster prototyping. It also studies if the flow is capable for generating production quality RTL targeting for ASIC.

Chapters 1 and 3 contain the required information for reader to understand the scope of the work. Chapter 1 concentrates on introducing general ASIC design flow and HLS flow. The time usage in both approaches is compared and RTL synthesis, verification, optimization and logic synthesis of HLS flow are shown. Chapter 3 presents FPGA prototyping technology, methods and benefits. Basic technology is introduced and an example of a FPGA and a FPGA prototyping environment are shown and they are later discussed in the work. Chapter 3 also presents FPGA prototyping flow and the benefits FPGA prototyping gives compared to the traditional ASIC design flow.

In chapter 4, a fully behavioral IP block is synthesized from the Simulink model to VHDL code. The chapter describes the steps to generate HDL code with HDL Coder–tool. Chapter 5 concentrates on verifying the generated VHDL in RTL simulator and in Chapter 6 the design is synthesized and the functionality is tested in FPGA environment. Chapter 7 introduces optimization techniques in the flow to generate good quality HDL code targeting for FPGA and ASIC.

Chapter 8 evaluates usability, performance and design flow time benefits that can be achieved by using the MathWorks HLS flow in SoC development. Finally, Chapter 9 summarizes the work.

2. HIGH-LEVEL SYNTHESIS

In this chapter, ASIC design flow and HLS are introduced. The main concentration is on the ASIC design flow times and complexity in today's commercial research and development. Principles of HLS are covered on those parts that are relevant for this work.

2.1. ASIC design flow

ASIC development from system specifications to silicon chip in telecommunications systems takes from a few months to a few years, depending on the application. Design flow generally consists of system analysis, coding and verifying a reference model of the system, coding and verifying a RTL model from the reference model, logic synthesis and physical fabrication. HLS improves the design flow by automating RTL generation from reference model combining it with logic synthesis. HLS flow and manual flow are presented in Figure 2. [5]
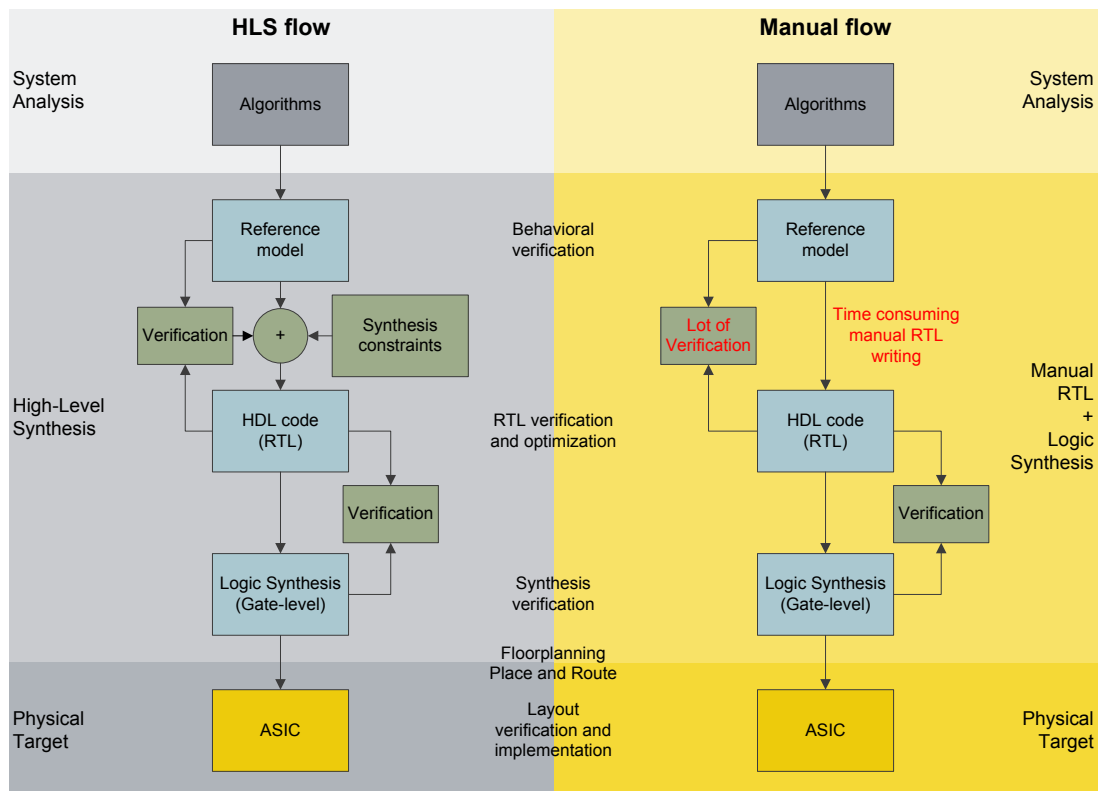

Figure 2. General design flow of ASIC development.

System analysis is an early design phase, which includes system specifications, architecture specification, coding and verifying algorithms of the system. Architecture and algorithm choices during the system analysis affect system development and may have a great effect on design complexity and performance of ASIC. Careful system

and architecture specification saves from increased design size, workload, design flow time and cost. [5]

General approach to create HDL code is to write a reference model of system based on algorithms and verify it in a simulator. The reference model describes the behavior of a system. RTL is hand-written based on the reference model and verified in the RTL simulator to have desired functionality. RTL is used to better describe HW functionality. Created HDL code is synthesized to gate-level model by logic synthesis and then further processed to a chip. [5][6]

Logic synthesis includes gate-level synthesis from RTL, floorplanning and Place and Route (PAR). Floorplan is done on fully functional design to place all the design blocks on silicon area. PAR is applied to create wire connections for the blocks and interface on block-level and top-level. After this, layout verification is done and when all the design specifications are met, a physical chip can be manufactured. [6]

HLS automates design from the reference model to synthesized netlist. Algorithms are first rewritten to the reference model then HLS is applied with synthesis constraints for the HDL code generation. The RTL simulator is used to simulate the generated HDL model and simulation results are verified to meet the desired functionality. A gate-level model is synthesized from the functional RTL model and verified to have the desired functionality by meeting timing and technology constraints. HLS includes automatic optimization methods to modify the RTL model to improve timing or area properties. [5]

HLS generates RTL code rapidly and it truncates design times. Verification of the RTL takes around 70 % of the whole design cycle so using HLS gives benefit in design flow times by allowing the verification to start earlier compared to the manual method [6]. HLS provides time benefit in iteration speed in case of a flawed algorithm model. It also moves the verification focus on algorithm and RTL verification becomes lighter. In HLS flow, changes made on the reference model are automatically changed in the HDL model after re-synthesizing the model. It provides an option to set the target FPGA for automatic target constraint setting. Time save in these phases improve the total flow time. Design flow timelines for manual and HLS ASIC development approaches are presented in Figure 3. [5]
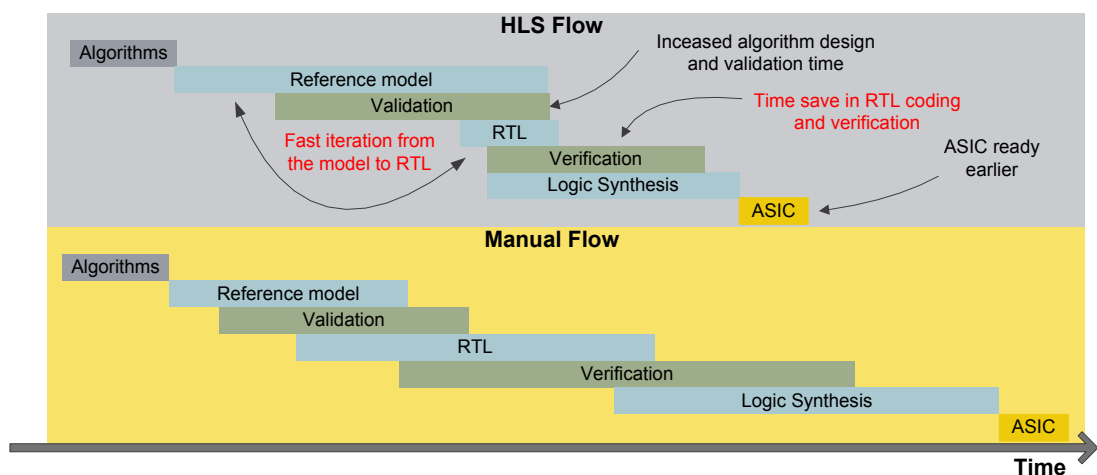
Figure 3. Approximate ASIC design flow timeline.

2.2.   Reference model to HDL code

ASIC design flow generally starts from algorithms. Based on the algorithms, a reference model is hand-written in high-level language, for example, SystemC, C++ or MATLAB. The reference model is behavioral model that requires no timing, concurrency or target technology information. [5]

A reference model is generally represented in floating point arithmetic. The floating point model is further converted to fixed-point arithmetic for synthesis. The fixed-point arithmetic enables usage of optimal word lengths and integral arithmetic on HW, therefore, is cheaper and smaller in area compared to the floating point arithmetic. Fixed-point model is verified in a simulator to have equal functionality. When the model has been verified and is working, synthesis constraints can be set. [5]

The synthesis constraints first specify the target technology and clock frequency. Then reset, clock enable behavior and process level handshake are introduced. Finally, individual constraints are set: I/Os, loops, storage and design resources. Synthesis flow is represented in Figure 4. [5]
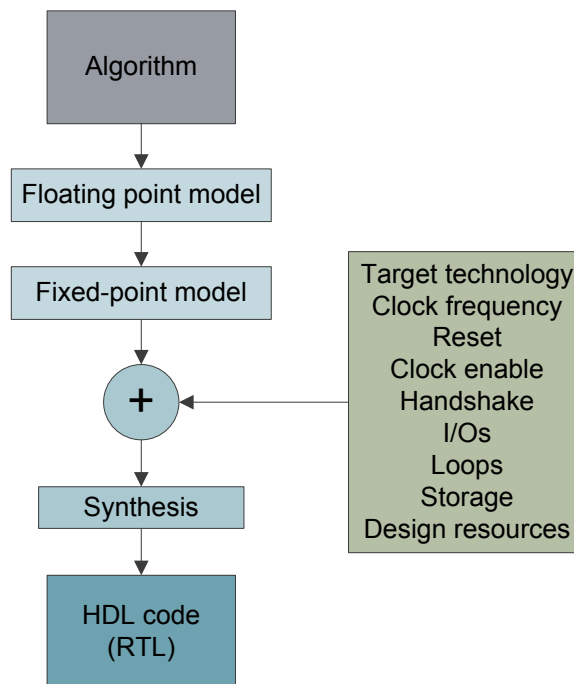
Figure 4. HDL code synthesis flow.

2.2.1. *RTL optimization*

System specification defines the goal for chip size, clock frequency and power efficiency. If requirements are not met, RTL can be optimized to reduce area, improve timing or lower power consumption of ASIC/FPGA. The RTL optimization is done after the functionality has been verified. [7][8][9]

Timing properties can be changed by tuning throughput, latency or local data path delay. The throughput means the amount of data that can be processed in a clock cycle and the unit is called bits per second (bps). The throughput can be improved by loop unrolling, which decreases the time between input reads. The loop unrolling decreases or eliminates loop control logic but adds more logic in the design, which increases the design size. Figure 5 below gives an example of the loop unrolling in case of calculating X in its 3rd power. [7]
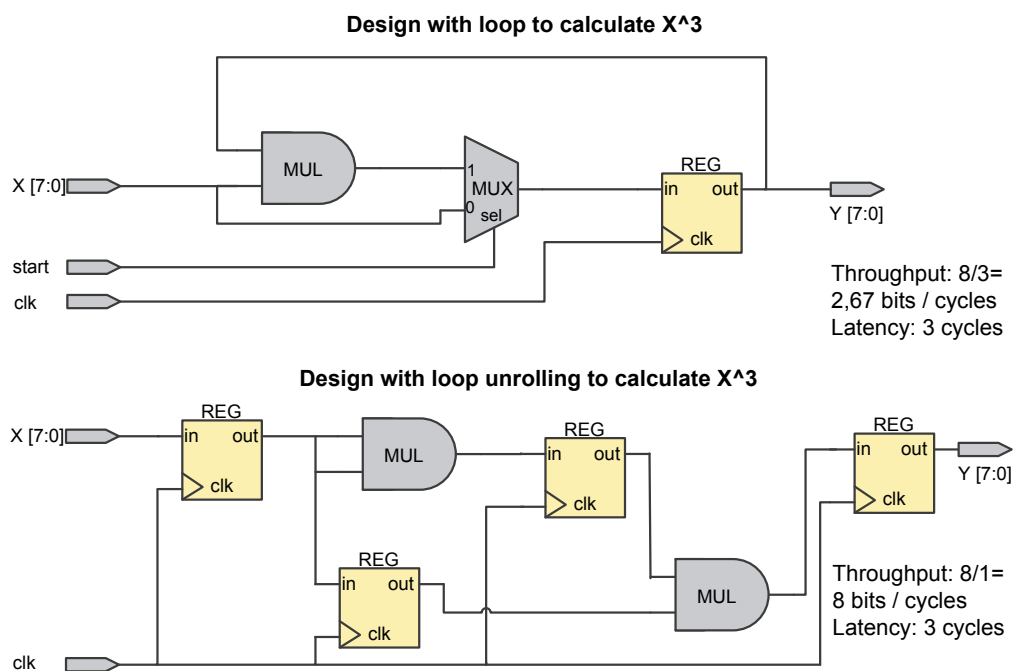
Figure 5. Loop unrolling in logic schematic view.

Latency describes the time it takes for data to pass from the input to the output of the circuit. It can be decreased by increasing parallelism and removing pipeline registers. Removing the pipeline registers increases critical path delay and decreases achievable maximum frequency. An example of removing pipeline registers is presented in Figure 6 below. [7]

Design to calculate X^3 with pipeline registers

Design to calculate X^3 without pipeline registers
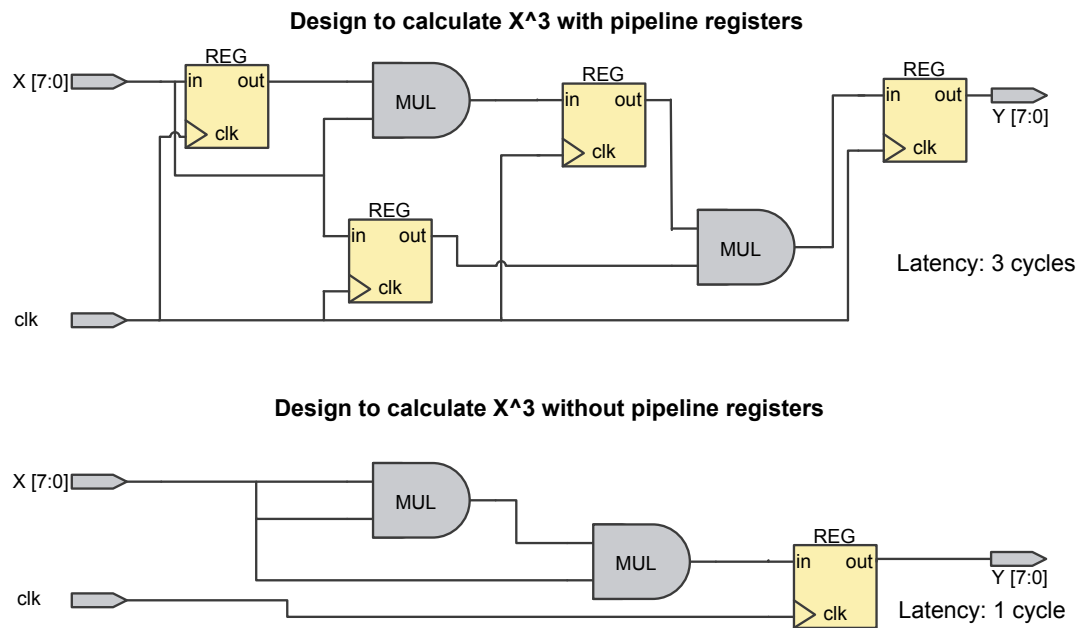
Figure 6. Pipeline removal in logic schematic view.

Logic data path delay is the time required for signal to pass through the logic between two sequential components. The maximum frequency of the circuit is limited by the local data path delay. To minimize the delay, more register layers can be added between the logic or register balancing can be applied. [7]

Area optimization can be done by reusing controllable logic, which is opposite to the loop unrolling. Adding multiplexers and control logic, for example finite-state machines (FSM), to the design decreases the amount of registers and arithmetic logic blocks. This further decreases the required chip area. [7]

Power optimization can be used to reduce the power dissipation of the circuit. The main reasons for the power dissipation are clocks of sequential circuits that are constantly switching. The clocks consume large part of the power in systems, up to 45 % of the whole power. One way to reduce the power consumption is to use clock-gating which decreases unnecessary clock switching for register that have no new input data. Another way to reduce the power consumption is to use sleep-mode optimization which shuts down multiplier when the output of it is not used. Both of these methods add logic in the circuit. Static power consumption can be reduced by using smaller power supply voltage and shutting off inactive parts of the system. [8][9][10][11]

2.3. Verification

Correctness of the ASIC design is the major focus point to avoid manufacturing costs of faulty designs and increasing time to market. RTL verification is more laborious than the reference model verification, therefore, it has to be done thoroughly to avoid re-spins. The verification is the most time consuming design phase in today's SoC development, taking approximately 70 % of the whole design flow time. [12]

The RTL verification requires testing the RTL design in every possible scenario to meet the functional specification. Therefore, it is not standardized for different designs. Increasing complexity of designs and non-standardized verification drives forward the IP reuse to speed up the design, the verification and time to market, and decrease the development costs. IP reuse means that a complex design is divided into smaller blocks, IPs. The IPs are verified blocks that can be effortlessly reused in other designs. [12]

The RTL verification includes lint checking, formal model checking, logic simulation, transaction-based verification and code coverage analysis. The verification is done first on IP level and finally on chip level. Verification flow is represented in Figure 7. [12]
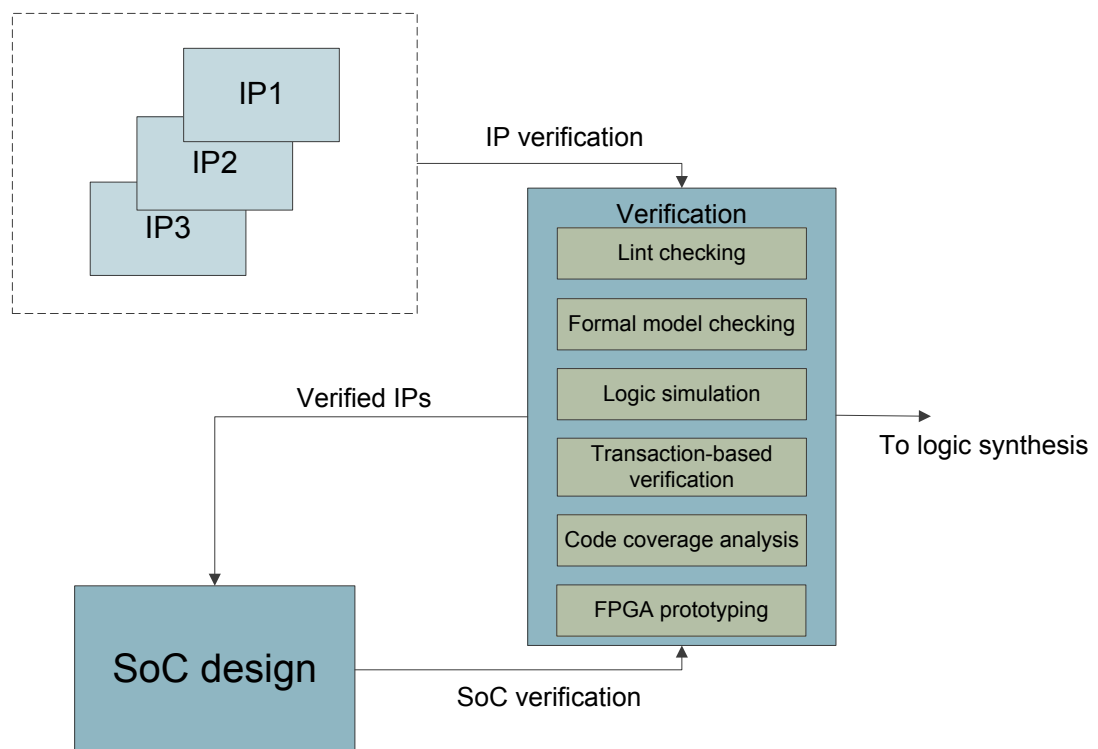
Figure 7. RTL verification flow in SoC development.

Lint checking is an early check to verify syntactical correctness of the code to prevent those errors to pass for more time-consuming, advanced tools. It reports uninitialized variables, unsupported constructs and port mismatches. [12]

Formal model checking compares system behavior to user-defined logical properties extracted directly from the design specification. For the verification, it uses mathematical methods and it works well for complex designs. [12]

Logic simulation can be done by two approaches; event-based simulator or cycle-based simulator. In the event-based simulator, the design is tested by one input stimulus at a time in chronological order. After the stimulus is given, it propagates trough the design and once steady-state condition is achieved again and new stimulus is sent. The event-based simulator is an accurate method to verify all the design elements but it is very time consuming on large designs. The cycle-based simulation works only on synchronous designs. It checks the logic between state elements and/or ports at once, therefore, each logic element is evaluated only once within a clock cycle. This makes the cycle-based simulation faster than the event-based simulation but vulnerable for simulation errors because it reacts only to the clock signal. [12]

Transaction-based verification allows transaction level simulation and debugging. It tests systematically every block level transaction of the system and it doesn't require detailed test benches. [12]

Code coverage analysis is performed to identify the untested areas of the design and provide an indirect measure of quality. It is performed on either block level or chip level RTL view and it lists untested or partially tested areas in the design. [12]

FPGA prototyping is a verification method that allows testing a design on HW against real-time I/Os and feedback. It enables early software (SW) development. FPGA prototyping is further introduced in Chapter 3.

2.4. Logic synthesis

Logic synthesis is used to compile the RTL design automatically into a gate-level netlist. The logic synthesis includes two phases; RTL read in phase and technology mapping phase. First, RTL is manipulated and combinational logic may be simplified depending on the RTL coding style. Next, after all changes to the combinational logic are made, the gate-level design is synthesized to match the RTL functionality with desired technology library. The libraries include different components so the gate-level design may vary depending on the technology library used. Simplified logic synthesis flow is represented in Figure 8. [13]

RTL design

```
entity MACHINE is
 port(
     A : in std_logic_vector(15 downto 0);
     B : in std_logic_vector(15 downto 0);
     C : in std_logic_vector(15 downto 0);
     D : out std_logic_vector(15 downto 0)
      );
end entity MACHINE;

architecture rtl of MACHINE is
begin
.
.
.
end  architecture MACHINE;
```

RTL manipulation

AB+AC => A(B+C)

Technology mapping

Gate-level design
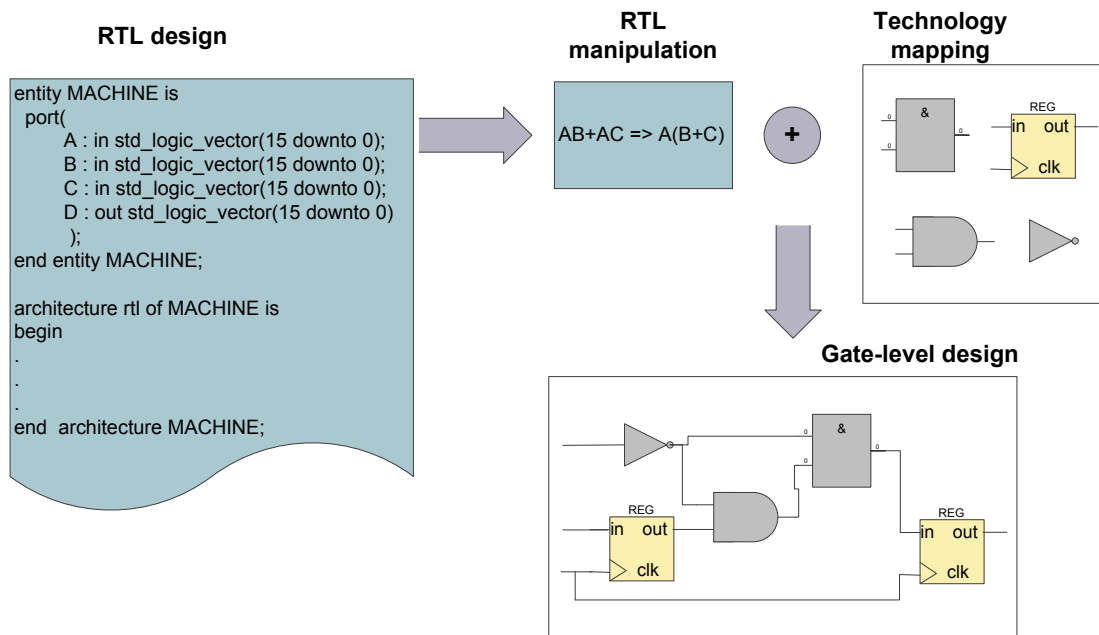
Figure 8. Simplified logic synthesis flow.

The logic synthesis is performed by a specific synthesis tools. Some of the tools are designed for FPGA synthesis and some of them are for ASIC synthesis. FPGAs have fixed resources and area, and also implementation is different compared to ASICs. Therefore, FPGA synthesis tools intend to utilize the fixed resources to achieve the user-defined performance goal. ASICs synthesis tools have no fixed resources, so the area optimization is more important. ASICs are generally running on higher frequencies than FPGAs, but timing optimization is essential in both cases. [13]

3. FPGA PROTOTYPING

In this chapter, FPGA-based prototyping is introduced. FPGA tools and technology used in FPGA prototyping, prototyping flow and benefits achieved by using FPGAs in large-scale SoC development are covered. Topics are introduced on the level that is necessary to understand the aim of the work.

3.1. FPGA technology and tools

FPGAs are reprogrammable silicon chips that provide hardware-timed speed and reliability. FPGAs have a matrix of Configurable Logic Blocks (CLB) connected through programmable interconnects and they can be reconfigured at any point of the design cycle. CLBs include logic gates, Look-Up Tables (LUT) and Flip-Flops (FF). Today's FPGAs also contain configurable embedded Static Random Access Memory (SRAM), high-speed transceivers and high-speed inputs and outputs (I/O). Therefore, they are an interesting solution in digital HW development. FPGA structure is presented in Figure 9 below. [14][15][16][17]
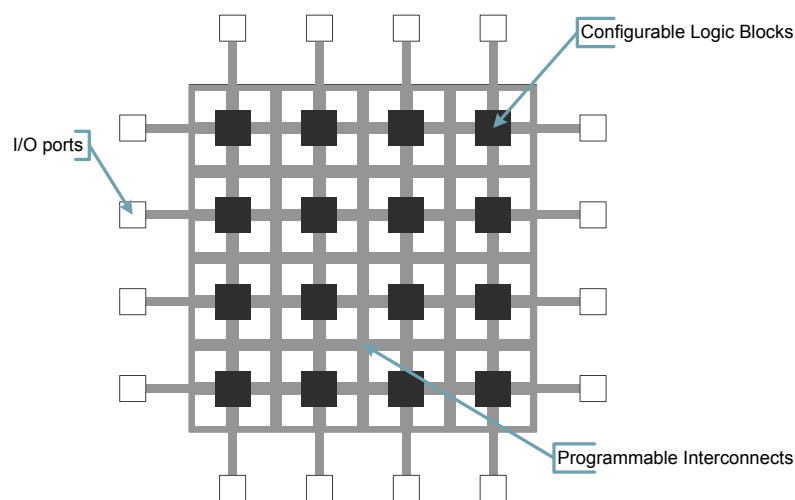
Figure 9. FPGA chip inner structure.

Two largest FPGA design and manufacturing companies today are Xilinx and Altera [18]. One example of a high performance FPGA family is Xilinx's Virtex-7 series. Features of the family are presented in Table 1 below. Virtex-7 FPGA will be discussed later in this work. [15]

Table 1. Features of Xilinx Virtex-7 FPGA family

Logic Cells	Block RAM (MB)	DSP Slices	Transceiver Count	Speed (GB/s)	Bandwith (GB/s)	Memory Interface (MB/s)	I/O Pins
2000000	68	3600	96	28,05	2,784	1,866	1200

To optimize the implementation different FPGA tools are used. The tools make good use of FPGA resources and it is equally important as the resources themselves. FPGA tools include synthesis, partitioning, PAR and debug tools. Comprehensive FPGA environments exist to centralize the FPGA prototyping. A FPGA prototyping environment includes FPGA, FPGA tools and required daughter boards. Electronic Design Automation (EDA) tools are used for similar purposes in case of ASIC. [17][19]

3.2. FPGA prototyping flow

FPGA prototyping flow consists of two branches; design and verification flow. The design flow includes HDL coding, the synthesis from RTL to the gate-level, implementation and FPGA programming. The verification flow includes functional simulations of RTL and the gate-level model, verification of implemented design and FPGA environment testing. Figure 10 represents the FPGA prototyping flow. [17][23]

The FPGA prototyping flow starts from creating the HDL code for the design and verifying the functionality in a RTL simulator. The code is generated for a FPGA test bench and might require some changes in clock and reset structures compared to ASIC code. The HDL code is synthesized into the gate-level model and it is formally verified to have the correct functionality compared to RTL. If the gate-level model is not fully functional, changes are made on the RTL code of the design and then it is re-synthesized into a new gate-level model. [17]

Once the gate-level model has the correct behavior, it is converted into a FPGA netlist. The netlist is further converted into a FPGA bit stream through technology mapping and PAR. The FPGA bit stream is verified in Static Timing Analysis (STA) and timing simulations. STA and the timing simulations are used to check that there are no timing violations in post PAR design in worst case Process, Voltage and Temperature (PVT) conditions. [17]

After verifying the FPGA bit stream, it can be programmed on FPGA. The design is tested in the FPGA environment with real-time inputs and feedback to verify that it is functional and behaving correctly with real-time I/Os. After verifying the functionality of the design in the FPGA environment, the ASIC design work towards ASIC optimized performance and physical fabrication can be started. [17]
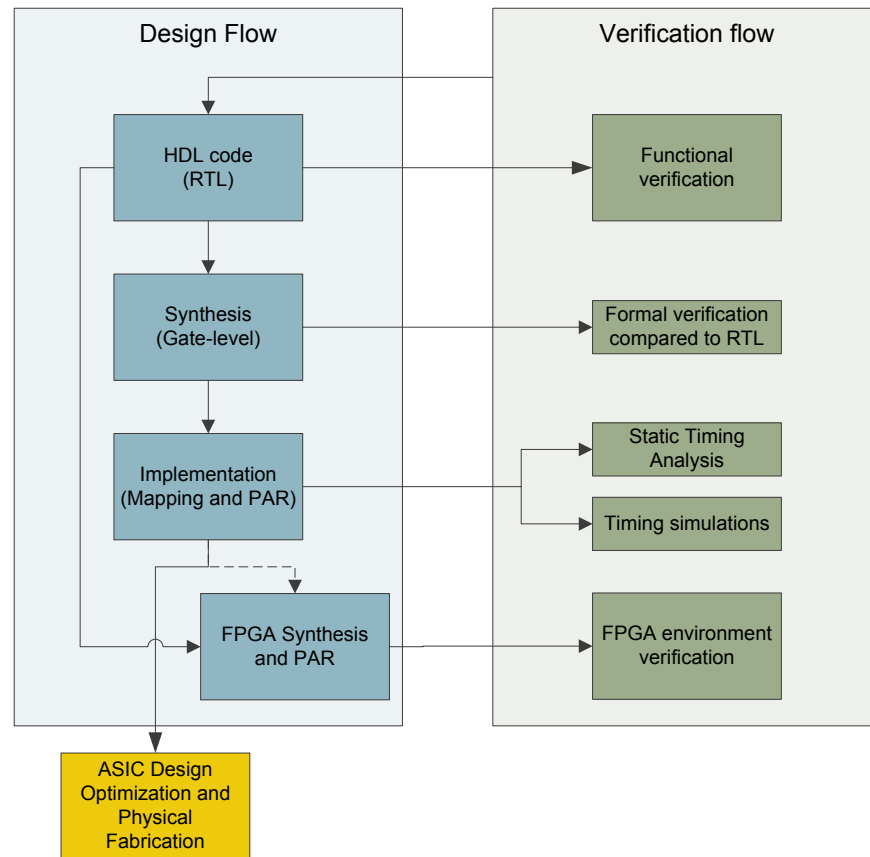
Figure 10. General FPGA prototyping flow.

3.3. FPGA prototyping benefits in SoC development

SoC verification is complex because its behavior depends on many variables: previous state, sequence of input signals and system effects of the SoC output, including the feedback. FPGA prototyping is a way to overcome these difficulties. It has a great advantage in pre-silicon verification over normal ASIC design flow by being the only testing environment that gives high performance and accuracy because of real-time dataflow, early SW testing and re-configurability. The FPGA prototyping improves the IP reuse and it may save from costly re-spins of flawed designs. FPGA are also getting faster so some of the designs might be prototyped on the same clock frequency as they are targeted to be on ASIC. Simplified timeline of FPGA prototyping benefits in HLS flow is presented in Figure 11. [17][23]
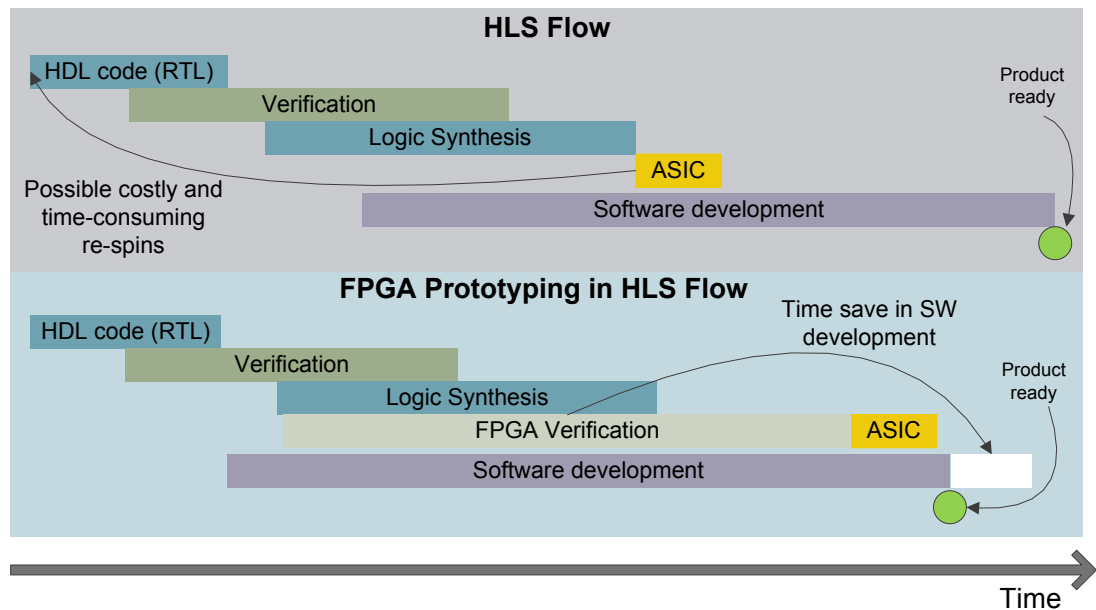
Figure 11. Simplified timeline presenting benefits of FPGA prototyping.

The real-time dataflow makes it possible to see immediate effects of real-time conditions, inputs and feedback on the system. Verifying the system in the real-time environment minimizes the possible flaws in the design and avoids from the costly ASIC re-spins. [17]

SW development is one of the major factors affecting the SoC development time. Testing of it can be started early in the design with specific SW testing tools but the tools have no real-time interface. The FPGA prototyping improves the SW development by enabling the testing of software in semi-real-time environment on FPGA to verify the SW functionality with real-world data. This shortens the SW development time after the chip fabrication and reduces time-to-market. Ease of re-configurability of a FPGA supports also both HW and SW development. [17]

The FPGA prototyping environment improves the IP reuse by enabling testing and verifying the functionality on the current design early on a FPGA. Using IPs generally reduces cycle time, cost and risk of the design. [23]

4. ALGORITHM DESIGN FOR HDL CODE GENERATION

MathWorks HDL coder is a HLS -tool to synthesize MATLAB or Simulink algorithm model to VHDL or SystemVerilog code. This chapter covers the design principles for synthesizable MATLAB or Simulink model. HDL synthesis is performed with an example IP block.

HDL coder uses the designed model, including the user-defined settings and the target technology files, as input to generate HDL code for both FPGA and ASIC. The tool has floating point to fixed-point converter built-in so both the floating point and the fixed-point algorithms are supported, which makes it flexible. However, HDL coder has some limitations on design principles to be able to synthesize the design into HDL. These limitations are discussed in the section 4.1 below.

4.1. MATLAB model

MATLAB is generally used for algorithm design of a system for fast simulation and verification purposes of the behavioral model. The models of telecommunication SoC IP blocks are generally large, which slows down the simulation. Therefore, algorithms are written in a way that maximizes the simulation speed. These algorithms may include processing large vectors of data at once and Object-Oriented Programming (OOP).

Since SW technology has more degrees of freedom compared to HW, HDL coder supports only a subset of MATLAB language that is targeted for HW. For example, synthesis from MATLAB OOP classes is not supported. However, it supports synthesis from MATLAB System Objects that are specialized objects designed for dynamic systems [24].

To produce rational HDL code, the algorithm should be written from the HW perspective. Algorithm models are often written into simulation optimized vector operations that create parallel structures and copies of combinational logic blocks in HW when processed by HDL coder. In real-time dynamic systems, input and output data varies over time, therefore, the system is not always required to process the whole data in one cycle. Loop structures can be automatically converted to streaming structures by using loop unrolling. However, parallel structures can be used if the target is to maximize the speed. To optimize the generated model, the algorithm model should be written in a way it is desired to be on HW.

An optimized way is to use only the necessary amount of the combinational logic to perform the logic operations within the timing constraint and multiplexing time-variant input signals into the circuit. This reduces the area of the hardware significantly as described in section 2.2.1. Optimization is further covered in Chapter 7.

The first thing when starting to design a model for the HDL code generation is to verify that data types, operators and control flow statements to be used are supported by the tool. These are presented in Table 2 below. [25]

Table 2. Supported data types, operators and control flow statements by HDL Coder [25]

Data type	Definitions
Integer	uint(8, 16, 32, 64), int(8, 16, 32, 64)
Real	double, single (for simulation and some high-end FPGA technologies supporting floating point data)
Complex	created by "complex()" -function
Character	char
Logical	logical
Fixed point	scaled, custom integer (max 128bits)
Vectors	unordered, row, column
Matrices	supported in the body of the design
Structures	supported in the body of the design
Enumerations	IP Core Generation, FPGA Turnkey, FPGA-in-the-loop, HDL Cosimulation
Arithmetic operators	
Binary addition	data type logical not supported
Matrix multiplication	
Arraywise multiplication	data type logical not supported
Matrix power	scalar types (exponent must be integer)
Arraywise power	scalar types (exponent must be integer)
Complex transpose	
Matrix transpose	
Matrix concat	
Matrix index	variables must be fully defined
Relational and logical operators	all common operators
Control Flow Statements	
For	no support for nonscalar expressions
If	no support for nonscalar expressions
Switch	uint(8, 16, 32), int(8, 16, 32), scalar

   To create synthesizable MATLAB code, the structure has to be correct. The design functionality has to be written in functions or MATLAB System Objects that are targeted for dynamic systems. Sub-functions or System Objects are then called within a main function to be included in the synthesis. Handshaking/synchronization between

blocks, functions, variable indexing and also signal buffering should be coded in the MATLAB design in a way it is desired to be in RTL.

Register modeling is done through "persistent" -variables. The variables that are wanted to save their states are defined in MATLAB function as persistent and these variables generate registers into RTL. In case of System Objects "static" –variables have the same behavior as "persistent" for MATLAB functions. HDL coder generates Read Only Memory (ROM) automatically into RTL from matrices and LUTs that exceed the user defined Random Access Memory (RAM) mapping threshold in the tool. Persistent array variables in the model are mapped to RAM by default to potentially reduce the area on the target device. The persistent array variables generate registers in RTL if they are not mapped to RAM.[25]

Generic variables cannot be trivially generated with HDL Coder. Lack of the generic variables may have negative influence in the IP reuse since the generated VHDL blocks are not easily scalable and have to be re-generated when signal bit widths or any scalable parameters are modified.

Two features of a model, that coding style has a great effect on, are speed and area. Essentially, increasing the area optimization decreases the speed and vice versa. This is not always the case but if the code is written rationally, it is a good rule of thumb. To minimize the area, it is desired to use as few arithmetic logic units (ALU) as possible, especially large multipliers or dividers due to their large size on chip. This method requires utilizing registers, multiplexers and control logic around ALUs to cover the desired functionality. This means dividing parallel operation in smaller pieces and looping these in sequences with fewer ALUs. The area optimization increases latency always and adds propagation delay in the design if slow combinational blocks such as multiplexers are used cover the parallel operations. However, the area optimization does not necessarily affect the functionality when the structures do not increase the critical path delay, therefore, become bottlenecks of the design. Figure 14 below illustrates the difference between the speed and area optimization with a simple multiplication circuit.

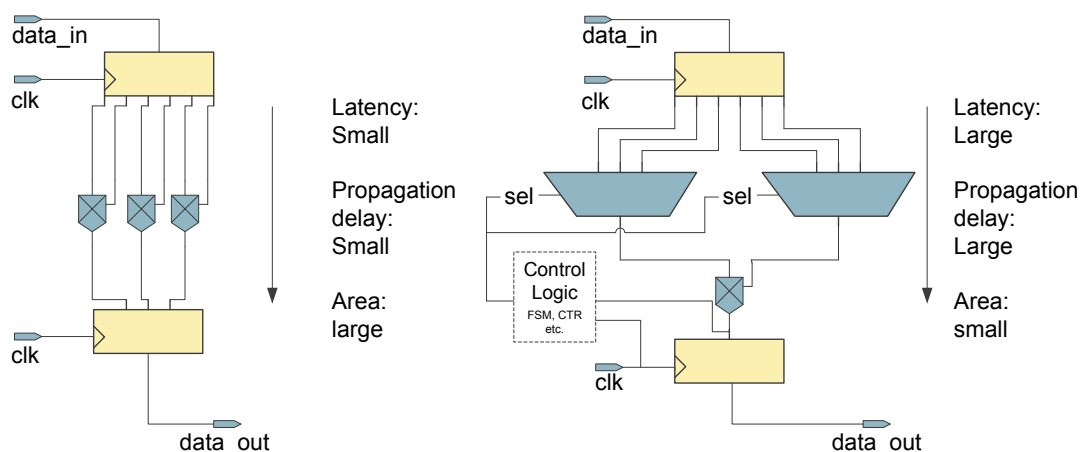Figure 12. A simple multiplication circuit in parallel and state controlled serial design.

When starting the coding, the target resources have to be known to be able to create a design that meets the requirements. This means that knowing the coding style that generates parallel or serial structure is essential. In MATLAB, parallel structure is generated if there is no state control utilizing arithmetic logic in a loop. This means

writing the arithmetic statements one below another. More area optimized structure is possible to achieve by using the loops together with the variable indexing and control flow statements to share the same ALUs with multiple operators. The variable indexing and control flow statements do not necessarily create serial structure and they can be used for parallel structure also. An example of parallel and serial structure in MATLAB algorithm is presented in Appendix 1. HDL coder is able to convert looping structures into streaming structures.

Hardware target, ASIC or FPGA, affects the model's area and speed optimization requirements. For ASIC design the general approach is to minimize the area of the chip by meeting the speed requirements. An FPGA, on the other hand, has fixed resources and the design is optimized to utilize all the available resources to maximize the performance. The FPGA resources are introduced in section 3.1. In this study, the IP block is targeted to be prototyped for FPGA environment testing but the final chip target is ASIC, therefore, both scenarios are taken into account.

The design of MATLAB algorithm block can be feed through type or can have registered output depending how output is assigned inside the block. If the output is assigned in the code before the functionality that manipulates the output's state, a register is automatically generated in the output of HDL version. If the output is assigned after the functionality, the block will have the feed through structure, thus increasing data path delay on the top level. This is important to take into account when writing larger designs where the path delays are critical. The example codes of these are represented in Table 3.

A feature that limits MATLAB for being used for large designs is that it has no concept of time. Therefore, it is not compatible with multi-rate designs using multiple clock domains.

4.2. Simulink model

Simulink is a graphical design tool that uses library blocks, MATLAB functions and System Objects to perform certain functionality. Simulink library includes vast selection of hardware optimized blocks and cover some of the functionalities needed in SoC development. User-defined MATLAB function blocks and System Object blocks can be used to create design specific functionalities or to reutilize existing MATLAB algorithms. Simulink supports the multi-rate designs, therefore, it has an advantage over MATLAB when a design has more than one clock domain. Simulink is used to create the example IP block in this work.

The example IP block is a system that scales and limits the power of IQ-data in a telecommunication SoC. The block is shown in Figure 13. The system is a multi-carrier system that processes IQ-data. It applies data scaling and power limitation carrier-wise. The design contains multiplication, addition, subtraction, multiplexing, state control, LUTs and accumulation, thus will create versatile HDL design.
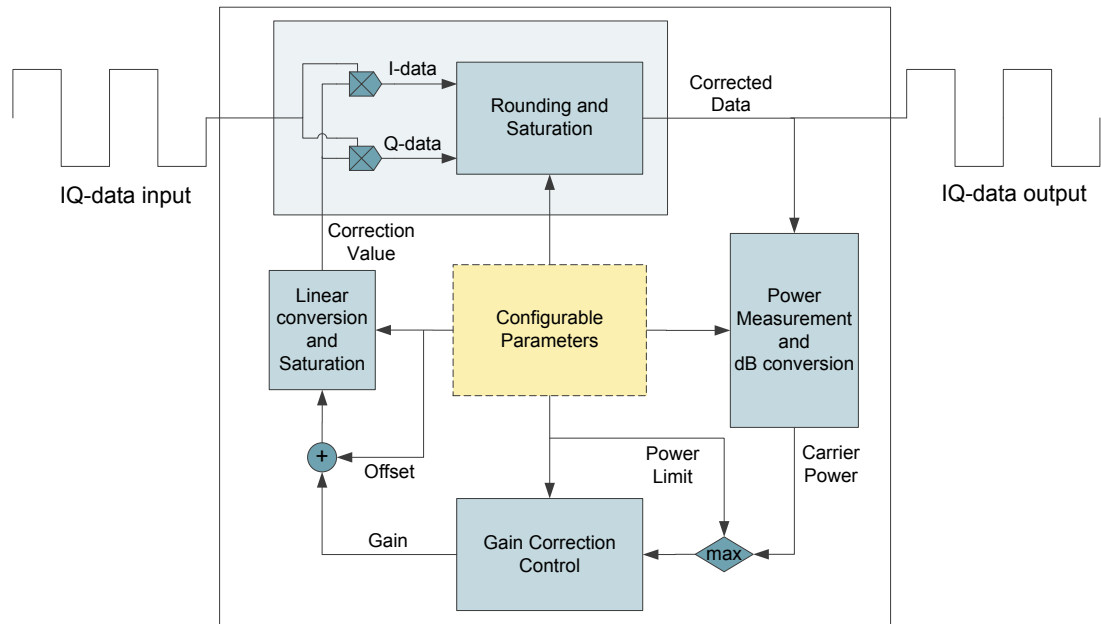
Figure 13. IQ-data scaling and power limitation system.

The IP block is built in Simulink by writing the algorithm with fixed-point data types in MATLAB function blocks and System Objects. The Simulink library components such as LUTs, delay and data type conversion blocks are used to complete the functionality. A test bench in Simulink is built by connecting all input variables to Device Under Test (DUT) and set their values manually or import them from the workspace. The data is imported as streaming data and the configuration parameters are constant values. In this example, the configuration parameters are set to limit the output below value 1. The test bench configuration is shown in Appendix 2 and an example simulation of the design is shown in Figure 14.
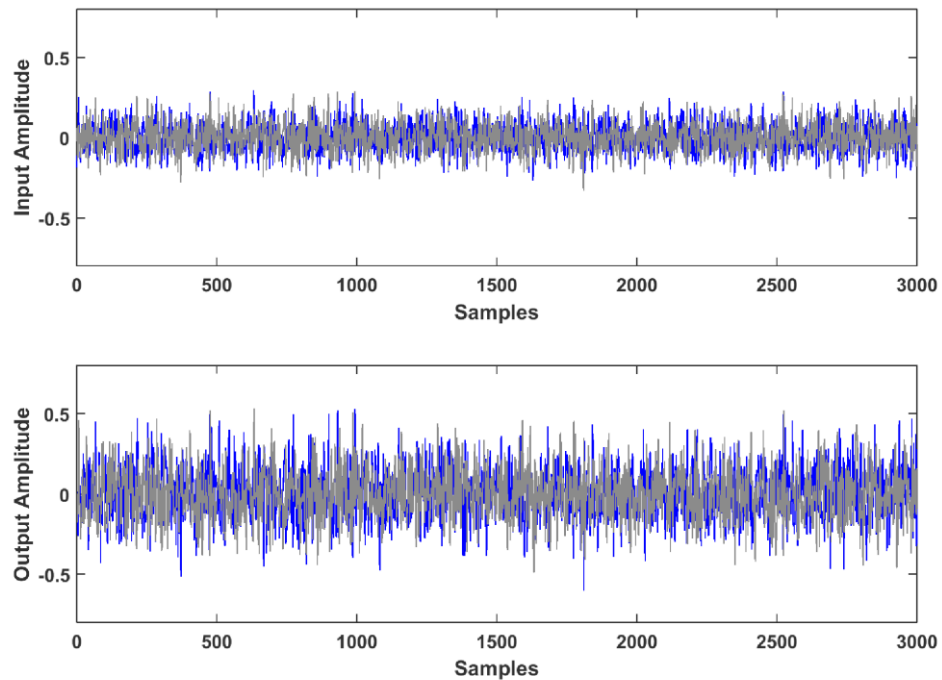
Figure 14. IQ-data input and output of power scaling block.

Creating hardware like design in Simulink using the default settings may increase simulation times compared to a simulation speed optimized MATLAB algorithm. However, Simulink includes acceleration methods to improve the simulation speed and partial quantization can also be used. Simulink has two acceleration methods, Accelerator and Rapid Accelerator, to improve the simulation times.

Accelerator mode generates and links code into MATLAB Executable (MEX) S-function written in C-language and uses this for the simulation. The code is stored for later simulations. In Accelerator mode, the target code methods are separate from Simulink software and MEX-files communicate with Simulink and MATLAB software via Application Programming Interface (API). The executable is run in the same process with Simulink and MATLAB. [26]

Rapid Accelerator mode differs from Accelerator mode by taking the solver with the target code methods to generate standalone executable located outside Simulink and MATLAB software. External mode is used to communicate with Simulink. Simulink and MATLAB are in one process and standalone executable is run on another processing core if available. [26]

The partial quantization defines only part of the design signals in fixed-point type and leaving the rest in floating point type. Defining only top-level inputs, outputs and coefficients in fixed-point format reveals roughly 80% quantization effects, thus is moderately good for verifying the algorithm behavior. Leaving sub-level components in floating point format improves the simulation speed compared to fully quantized model because the floating point data is lighter to process for MATLAB. The partial quantization could be used instead of scaled integer types as some of the simulation optimized algorithms use. Furthermore, it might result in even faster simulation times due to removal of scaling operations of the data.

For HDL code generation Fixed-point Converter can be used to convert floating point data types into user-defined fixed-point types. Solver setup can also affect the

simulation speed. "MultiTasking" option can be used to speed up the simulation but it's not supported for HDL generation. For HDL code generation "SingleTasking" option has to be used.

4.3. HDL code generation from model

Completed and behaviorally verified algorithm design can be synthesized into RTL. HDL coder takes MATLAB or Simulink design as input and generates either VHDL or Verilog from it. In Simulink case all blocks inside the top-level design are synthesized into separate VHDL-files and are imported as components on the top-level VHDL. An example of the code generation principle is shown in Figure 15. The top-level VHDL-file includes design I/O ports, internal signal declaration, port mapping of the components and assigning internal signals into the I/O ports.
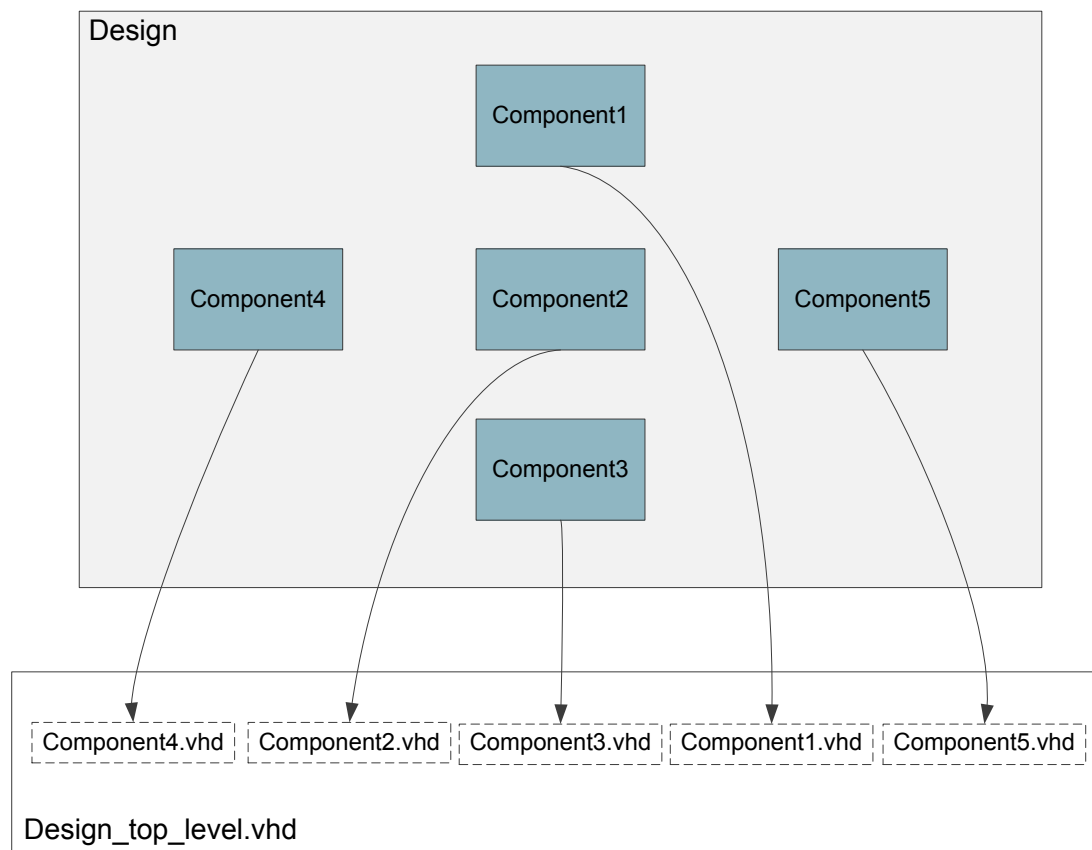
Figure 15. HDL code generation principle from Simulink/MATLAB Design.

Generating HDL code from a MATLAB design follows the same principle as shown above if sub-functions are written in separate MATLAB function files and are called within a main function. If all the functionality is written in a single MATLAB function or System Object, the whole design is synthesized into a single VHDL-file.

The target specific parameters described in section 2.2 can be configured by the user in HDL coder. After setting the parameters, HDL coder performs checks for global

settings, algebraic loops, compatibility and sample time to verify that the design is synthesizable. Depending on the design size, the RTL synthesis takes from a few seconds to a few minutes. The output is human readable HDL code making it really interesting from the point of RTL coding by automating the RTL code generation. The design flow times are introduced in section 2.1. If changes are made on the algorithm design, a new HDL code can be rapidly generated. The generated HDL code also provides visibility backwards to the algorithm model from VHDL-file trough links that take the user to corresponding MATLAB function. It also preserves all the comments of the MATLAB function into VHDL. The example MATLAB code and generated VHDL code example are shown in Appendix 3.

HDL coder generates traceability, resource utilization, critical path and optimization reports for the RTL model automatically but user can also disable the report generation. In Figure 16, is a high-level resource report of the example design. The user can also view detailed resources block by block. The critical path of the VHDL design can be back annotated in Simulink model and this is presented in Figure 17.

Multipliers	6
Adders/Subtractors	67
Registers	214
RAMs	0
Multiplexers	92

Figure 16. High level resource report of the example design.
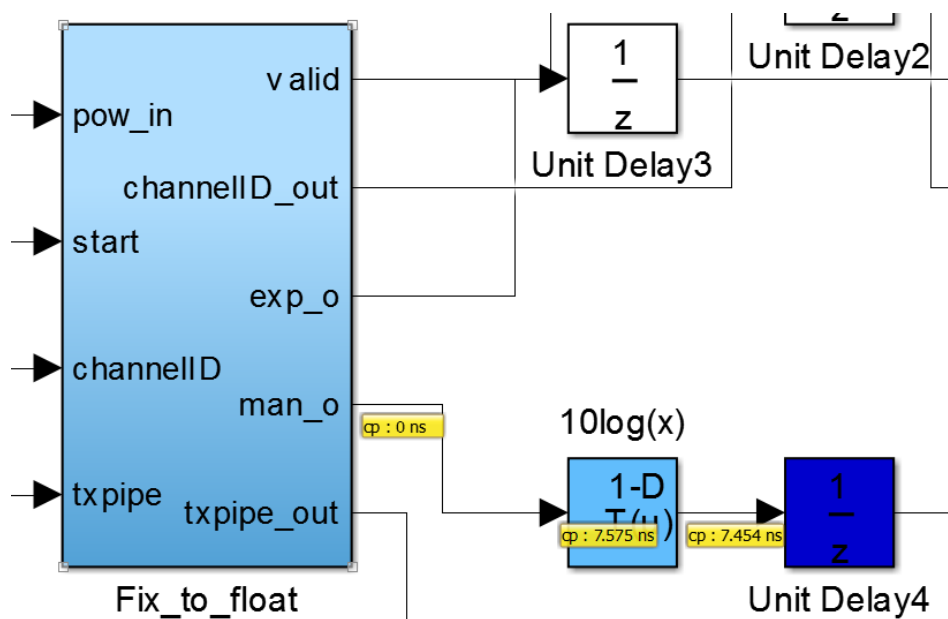
Figure 17. Critical path of the VHDL design highlighted in Simulink model.

Table 3 below presents good design practices and coding styles to create synthesizable HDL. The table includes the findings done during this work and might not include all the coding rules or methods to improve the design flow.

Table 3. General coding and simulation rules and methods for HDL code generation

Coding /simulation style or method	Description	Code example
Avoid using normal MATLAB Objects	No support by HDL Coder. System Objects can be used to create similar structures.	
Use streaming data	Streaming should be used, when possible, except if the data to be processed is vectorized	

One main function and call sub-functions inside	To include all the functionality in HDL every function should be called inside the same main function.	%main function
		function main(x, y, z)
		  %call sub-function #1
		  out1 = sub1(x)
		  %call sub-function #2
		  out2 = sub2(y, z)
		end

To create registers into design use "persistent" - variables	Defining a variable as persistent generates a register from it. For example, data buffering or value storage. Matrices and LUTs that exceed user-defined RAM mapping threshold are mapped to ROM. Persistent array variables are mapped to RAM by default.	%define variable or array
		persistent value_reg;
		%define the type e.g.
		%fixed-point variable
		%signed with one integer bit
		%and one fraction bit
		value_reg=fi(0,1,3,1);
		%array of four fixed-point
		%variables
		value_reg=fi(zeros(1,4),1,3,1);

Use at least one delay block in a Simulink feedback loop	HDL Coder requires at least one delay block in feedback loop for HDL code generation.	
Use Simulink for multi-rate systems	MATLAB supports only one clock rate in the system but Simulink supports multiple rates.	

Use state controlled structures and loops around arithmetic functions to minimize area	State controlled structures create multiplexers and registers, but utilize little arithmetic logic. Decreases design speed.	if (start == true)
		  for i = 1:4
		    mul(i) = x(i)*y(i);
		  end
		end

Use parallel arithmetic operations to increase design speed	Parallel structures minimize delays, but increase design size on chip.	if (start == true)
		  mul(1) = x(1)*y(1);
		  mul(2) = x(2)*y(2);
		  mul(3) = x(3)*y(3);
		  mul(4) = x(4)*y(4);
		end

Use LUTs for complex arithmetic operations	LUTs are an area and speed efficient way to replace complex and large arithmetic logic when the range of values is known. For example, logarithmic operations.	Different types of LUTs can be found from Simulink library

Add +1 to MATLAB indexes to utilize same test bench for RTL simulation	MATLAB indexes start from 1, but VHDL indexes start from 0 so user should add +1 to MATLAB indexes inside a model to be able to utilize same test bench with RTL model (If test bench indexes are set to start from 0). Compiler automatically removes the "+1" from the HDL code.	if (ct(indx+1) < limit)
		  mul = x(indx+1)*y;
		end

Assign values into output before operation to create output register	If a value is assigned into output before actual operation that manipulates signal value, HDL coder creates a register in output.	output = mul_reg;
		if (start == true)
		  mul_reg = x*y;
		end

Assign values into output after operation to create feed through structure	If a value is assigned into output after actual operation that manipulates signal value, HDL coder generates feed through structure in output.	if (start == true)
		  mul_reg = x*y;
		end
		output = mul_reg;

Use signal specification block in Simulink to define data type in a feedback loop utilizing signal type inheritance	If Simulink gives an error for detecting incorrect data type, use signal specification block to force data type. This can occur when signal type is inherited from previous blocks in a feedback loop. It doesn't create any additional HDL code.	
Use floating point data types for faster simulation	Floating point data types are faster to simulate and can be converted automatically to fixed-point data types by Fixed-Point Converter.	

Use Simulink simulation accelerators to improve simulation speed	Accelerators in Simulink utilize MEX-files to separate target code from Simulink software and run them separately but communicating with Simulink/MATLAB trough API. This provides improvement in simulation time.	
Use "single tasking mode" in Simulink for HDL code generation	"Multi tasking" mode can be used to improve simulation speed, but HDL Coder requires single tasking mode.	

5. HDL CODE VERIFICATION

In this chapter, HDL code verification possibilities that the MathWorks workflow provides are discussed. General SoC verification flow is shown in section 2.3. HDL Coder provides two options for RTL simulation; co-simulation and RTL test bench generation.

5.1. Verification in RTL simulator

Co-simulation automatically generates stimulus for a HDL model from MATLAB/Simulink test bench and runs a RTL simulator in the background. Co-simulation compares output of the code generation model of the algorithm to the HDL model's output. HDL Verifier is required to be installed. The HDL model is simulated in the background in user-defined RTL Simulator and the output is imported in MATLAB. Co-simulation compares the models bit-accurately and cycle-accurately. Simulation configuration is presented in Figure 18 below. [27]

MATLAB

| Output Arguments | ← Stimulus | MATLAB test bench MEX-function | Response ← | Input Arguments |

HDL Simulator

In

VHDL design

Out
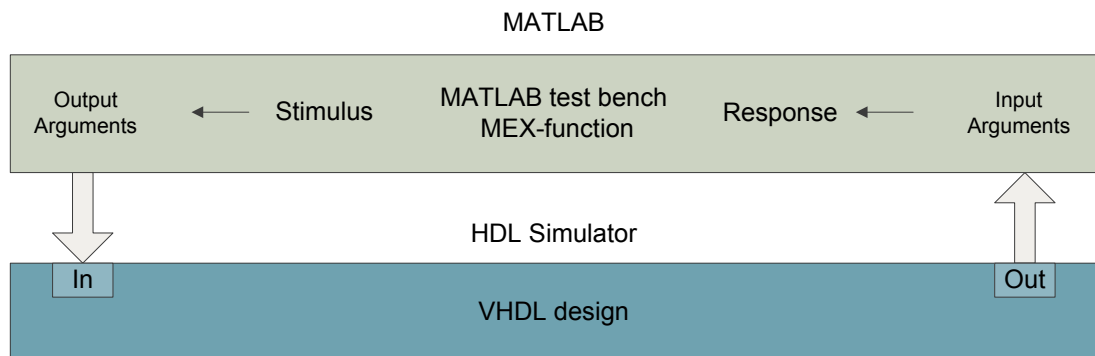
Figure 18. HDL co-simulation configuration.

In co-simulation, configuration MATLAB functions as a server and HDL Simulator as a client. MATLAB/Simulink test bench signals are connected to the VHDL design's input ports and the design's output ports are connected back to MATLAB with proper arguments. Test bench MEX-function feeds the HDL Simulator with the stimulus from the MATLAB/Simulink test bench and receives the response from the VHDL design.[27]

Error is calculated from the differences of code generation model simulation and RTL simulation. The error comes mostly from quantization inaccuracies and if the algorithm model is written with fixed-point data types the error should be zero. The comparison is done on the output ports and it is a rapid way to verify HDL design correctness every time the algorithm is changed. Co-simulation window of the example design is presented in Figure 19.
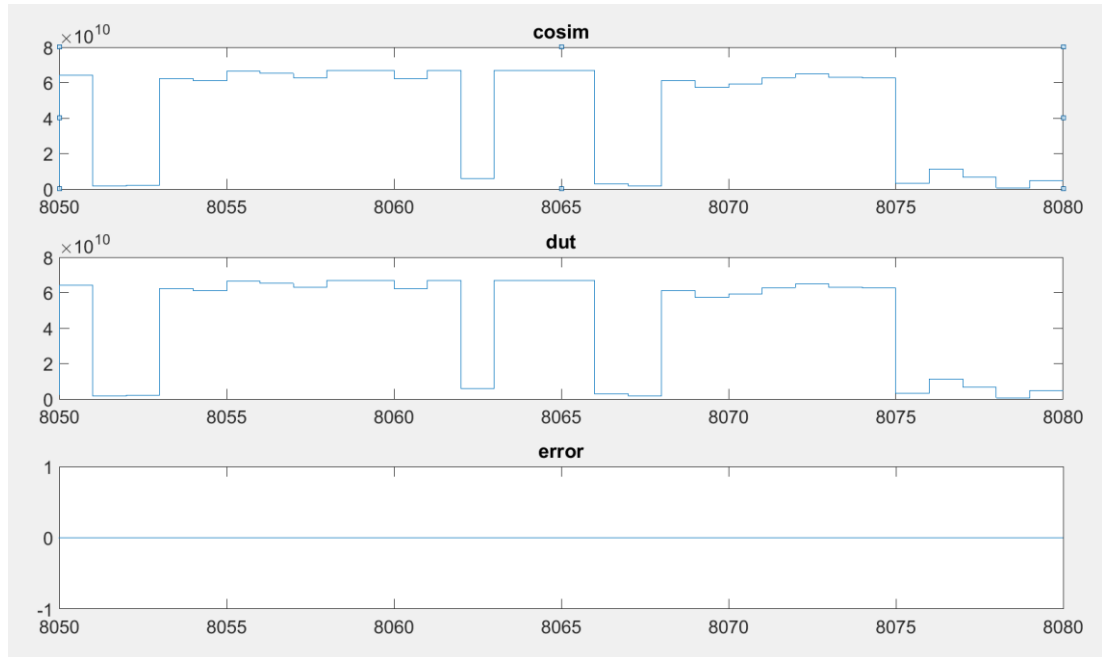
Figure 19. Co-simulation window where error between Algorithm model and HDL model is shown.

The other way to simulate RTL is to generate a HDL test bench with HDL coder from MATLAB/Simulink test bench and simulate it manually in a RTL simulator with the generated HDL design files and HDL test bench. Generating the HDL test bench takes roughly two times longer than the entire co-simulation, thus the user have to manually verify the correctness of design functionality in the RTL simulator. Manual verification further increases the verification time described in section 2.3. The automatic HDL code generation from the algorithm model and the RTL functional verification against the algorithm can provide great improvement in prototyping times and efficiency. It also moves the verification focus on the algorithm model. Therefore, it is suitable for the SoC prototyping purposes.

5.2. Additional RTL verification methods

HDL Coder has support also for other RTL verification methods such as lint checking, code-coverage analysis and verification with validation model described in section 2.3.

The code-coverage analysis is done on the algorithm model and since HDL Coder generates the RTL from the algorithm there is no need for RTL code-coverage. The code-coverage checks that all the functions defined are used, all statements are executed, all branches are executed at some condition and all Boolean expressions are evaluated to true and false. This is a fast way to check if there are some functions that are not executed in any conditions.

HDL Coder supports 3rd party lint checking tools. These are Ascent Lint, HDL Designer, Leda and SpyGlass. By enabling the lint checking, the tool generates a script

for specified lint tool and user can also configure lint checking parameters to meet the requirements. The lint checking is used to check suspicious behavior of the model such as division by zero or assigning values to a variable before variable declaration. [25]

HDL Coder provides validation model verification method to verify functional equivalence of the original algorithm and the code generation model. Difference to the co-simulation is that this compares the original model to the code generation model over comparing the code generation model to the RTL model. Both models are fed with the same stimulus on each time step and output is compared similarly. The example design's validation model simulation output is shown in Figure 20 below.
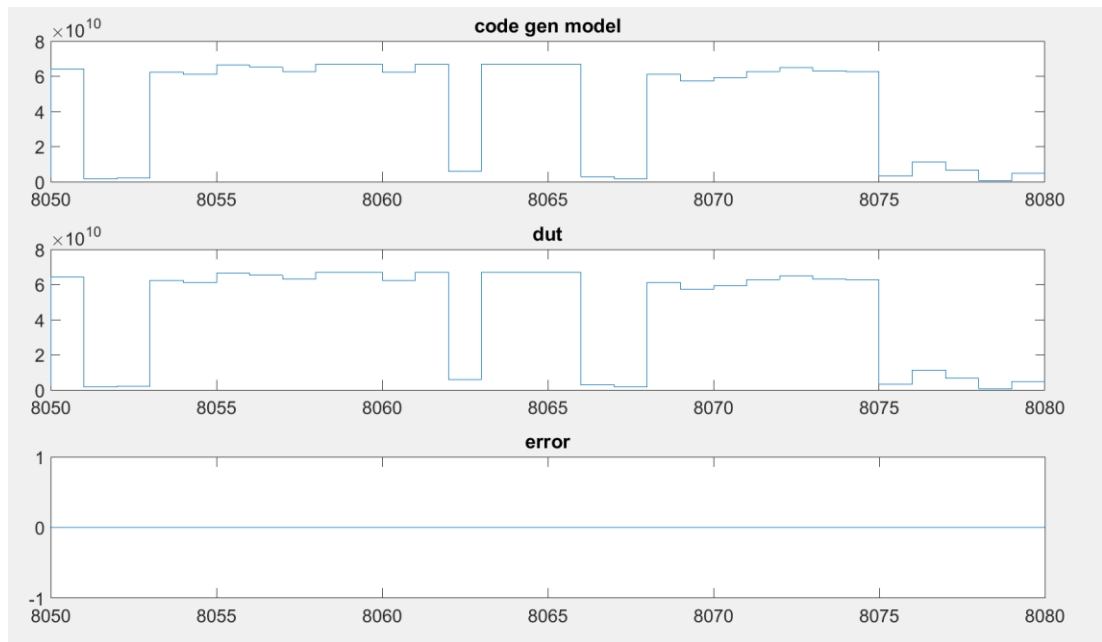
Figure 20. Validation model simulation output.

The flow has vast support for different RTL verification methods and all of them can be controlled within one tool. This provides improvement in prototyping flow clarity and may slightly improve the design flow times by automating the 3rd party tool usage. For very detailed verifications with 3rd party programs, it is easier to use the tools manually with Graphical User Interface (GUI) due to better visibility to the configuration parameters.

6. FPGA SYNTHESIS AND FUNCTIONAL VERIFICATION IN FPGA ENVIRONMENT

In this chapter, VHDL model is first synthesized to a gate-level design and further synthesized into a FPGA programmable model. Synthesis results of the hand-written and the HDL coder generated VHDL are compared. Furthermore, the design is programmed on FPGA and the functionality is verified. FPGA structure, prototyping and verification were introduced in sections 3.1 and 3.2.

6.1. Logic synthesis and comparison

The logic synthesis, introduced in section 2.4, is performed after the HDL model has proper functionality. The flow does not include own synthesis tool but it supports the following tools: Xilinx ISE, Xilinx Vivado, Synopsys Synplify Pro, Altera Quartus II, Mentor Graphics Precision and Microsemi Libero. The user can choose to use any of the listed tools depending on the requirements. HDL coder generates a tool specific script which is used to start the selected tool and synthesize the generated RTL code with the user-defined settings. Synthesis time and result depends on the VHDL model and the chosen synthesis tool. In this work, Xilinx Vivado was used to synthesize the hand-written model and the HDL coder generated model of the example IP. Hardware resource utilization results are shown in Table 4 below.

Table 4. Hardware resource utilization results of the hand-written and the HDL coder generated IP

	HDL Coder generated IP (targeting FPGA) resource utilization compared to hand-written IP
Flip-Flops	81,1 %
LUTs	54,6 %
Memory LUTs	60,9 %
I/Os	54,0 %
Block RAMs	16,7 %
DSP48s	100 %
Clock Buffers	100 %

The algorithm model used for the HDL code generation has roughly 80 % of similar or identical functionality of the hand-written IP so the synthesis results cannot be compared accurately. However, as can be seen from Table 4, The HDL coder generated RTL utilizes less resources than the hand-written model. Synthesis tools provide also an area report presented in logic cells. The synthesized model uses 61,7 % of the area of the hand-written model, thus seems to follow similar trend with the resource utilization report. Therefore, it is beneficial for FPGA based rapid prototyping due to faster iteration times compared to hand-writing by automating HDL code generation. It also creates synthesizable HDL and logic in reasonable size.

Maximum frequency for design was also compared and derived from the critical path delays received from the synthesis. The critical path delay for the hand-written model was 20,198 ns, including 2,597 ns of logic delay and 17,601 ns of route delay. The critical path delay for HDL Coder generated model was 7,781 ns, including 4,382 ns of logic delay and 3,399 ns of route delay.

The maximum frequency was derived from these values using the following equation

$$f_{max} = \frac{1}{\tau_{critical}} \tag{1}$$

where $\tau_{critical}$ is critical path delay.

The hand-written model is able to run at 49,5 MHz and the generated model at 128,5 MHz. The hand-written model was targeted on ASIC and the generated model on FPGA so the maximum frequencies are not fully qualified to be used for comparison. However, it can be said that by following the good algorithm coding rules rather good design speed can be achieved with the HDL Coder workflow.

6.2. FPGA environment verification

FPGA verification was done with FPGA-in-the-loop (FIL) configuration using the generated design and Altera Arria V Development Kit. Altera Development Kit utilizes two Arria V GT FPGAs and all the common I/O interfaces required for FPGA development. It provides a sufficient platform for testing IPs but might be slightly light for system level testing. High-performance prototyping environment would be better for prototyping larger SoCs and it is discussed in section 8.3.1.

The FIL flow performs the whole FPGA prototyping flow introduced insection 3.2 and it provides capability of using MATLAB or Simulink for testing the design in a real hardware environment. After HDL code generation it performs logic synthesis and generates a FPGA programming file with target FPGA specific files. FPGA is further programmed through GUI through either Ethernet or JTAG connection. The FPGA programming file can also be generated from the hand-written RTL by using FIL wizard [27]. In this example, logic synthesis was done by using Altera Quartus II.

Programmed FPGA is running in real hardware environment with MATLAB or Simulink stimulus. Data is streamed through FPGA chip and output is compared to the algorithm simulation output. FIL configuration is shown in Figure 21.
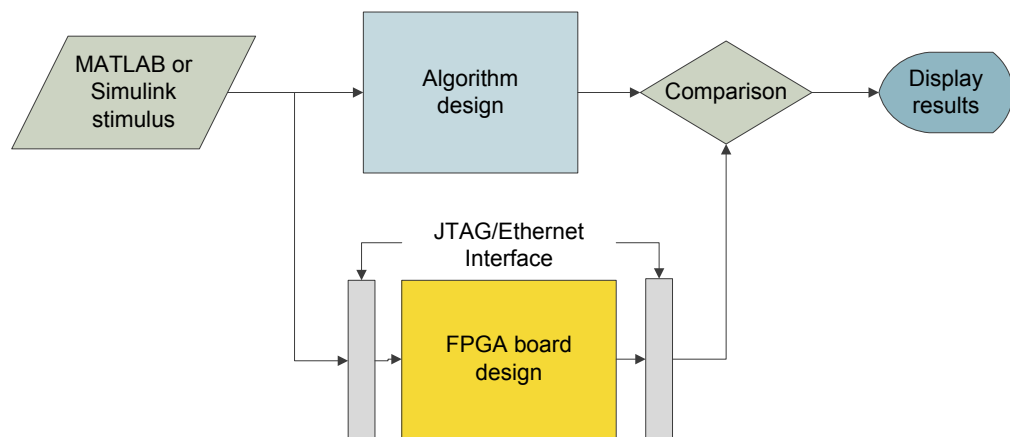
Figure 21. FPGA-in-the-loop simulation configuration.

The FIL output is a similar window as in co-simulation in chapter 5.1. The output data from the FPGA board and the algorithm is presented as waves and the difference between the outputs is compared in an error plot. The FIL simulation window is shown in Figure 22. In this example JTAG connection was used for data streaming. For larger simulations Ethernet is better to use instead of JTAG for higher data rate.
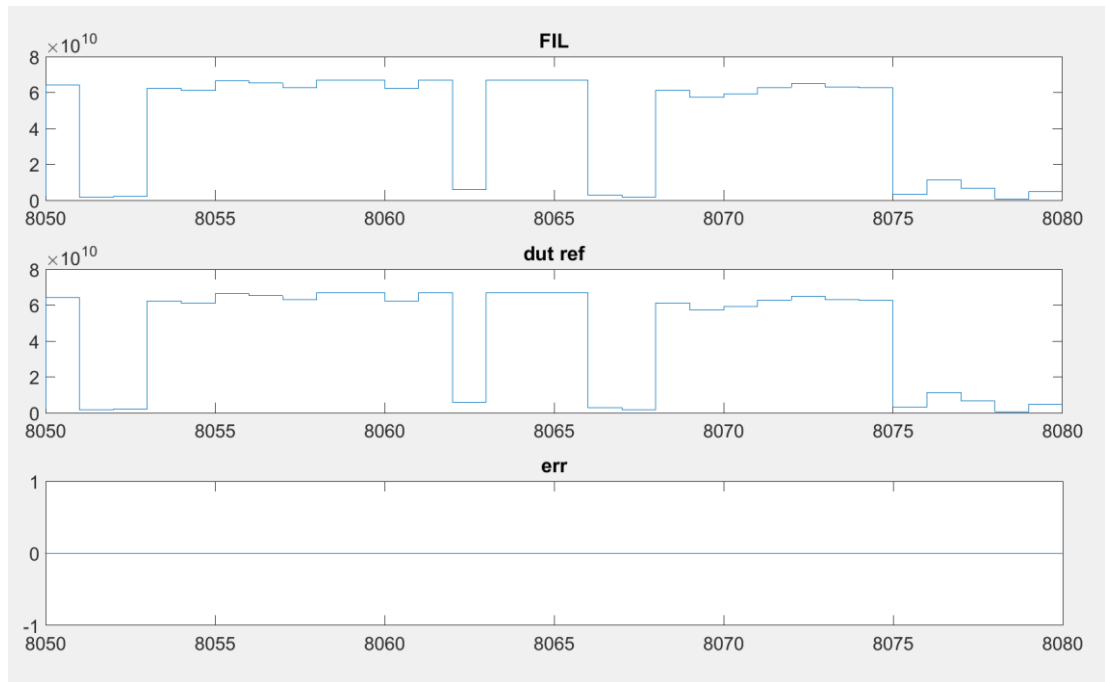
Figure 22. FPGA-in-the-loop simulation window presenting error between DUT and FPGA.

From the simulation results in Figure 22, it can be seen that the FPGA model has the same functionality as the algorithm model created in the beginning of the work. The results verify that the FIL flow produces improvement for rapid IP prototyping compared to manual verification in FPGA environment by decreasing the verification times and also making the prototyping flow faster and easier from algorithm into FPGA prototype. The flow provides also additional target workflows such as Generic ASIC/FPGA, FPGA Turnkey, Simulink Real-Time FPGA I/O and IP Core Generation. FPGA-in-the-loop was the only workflow used in this work.

7. HDL CODE OPTIMIZATION

HDL Coder provides optimization features that user can apply on a design. Optimization features include adding pipeline registers, resource sharing and loop unrolling introduced in section 2.2.1. In this study, optimization was done in algorithm. Tool configurable optimization features were tested on the example design but were not taken into use.

7.1. Optimization for FPGA target

HDL coder allows user to specify optimization features on top-level or on a single block. HDL Properties window allows the user to set input and output pipeline register count, sharing factor and streaming factor. Setting "Distributed Pipelining" option "on" lets HDL Coder to distribute existing or added pipeline registers across the selected block to improve the timing characteristics. "Constrained Output Pipeline" count can be set to redistribute existing delays within your design to meet the constraints. Registers specified by "Constrained Output Pipeline" are not affected by "Distributed Pipelining". RAM mapping can be also used to map registers on RAM to save area. It can be specified on every block separately and it only maps those registers to RAM that are larger than the threshold value. HDL properties window is presented in Figure 23 below.

Figure 23. HDL Properties window to set optimization parameters.

Using built-in optimization features efficiently requires understanding the design and the hardware implementation. Adding unnecessary pipeline registers increases circuit delay and using 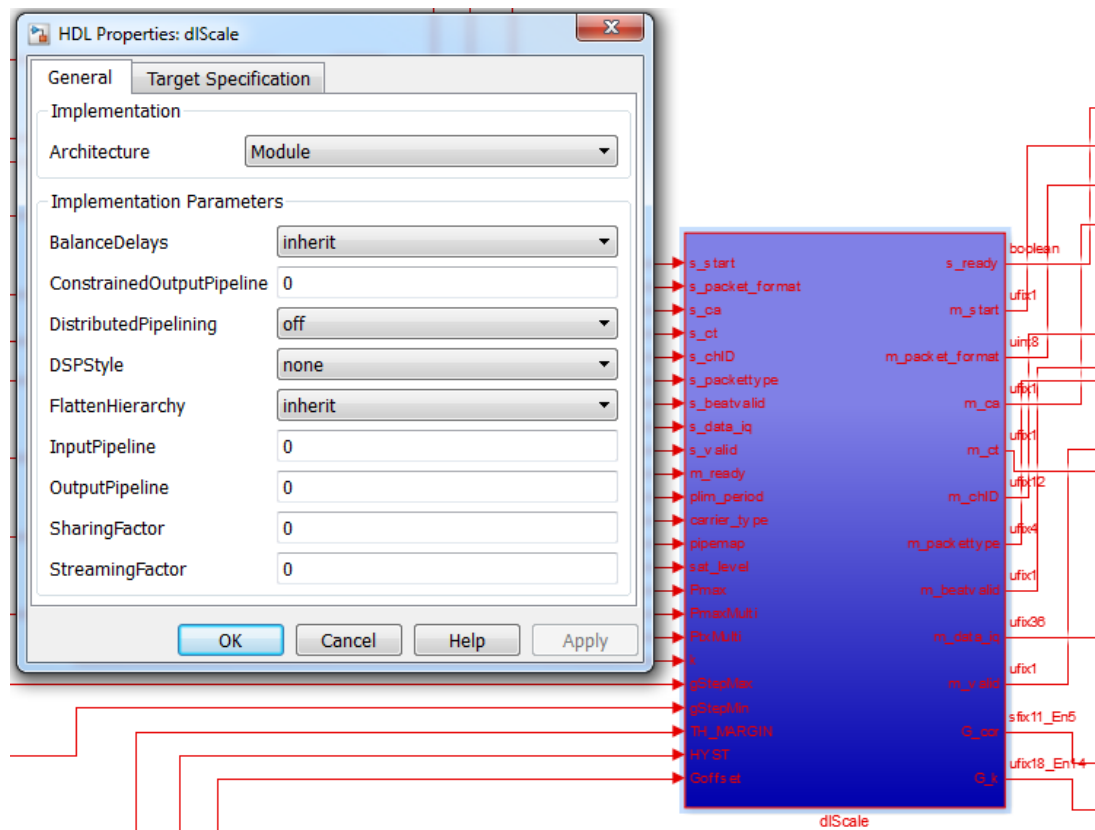sharing or streaming options on already optimal blocks may increase the design complexity and decrease the design quality. Therefore, if algorithm is designed with optimized functions, HDL Coder's optimization features have no effect but may reduce the quality. In this work, the example IP was already optimized on function level, therefore, no optimization features were used. An example of bad RAM optimization result is shown below.

For example, persistent variables configured in a "for"-loop including nested conditional statements utilized more resources when RAM mapping was enabled than without. Illustration of this structure is shown in Appendix 4. Using this kind of structure uses more than one clock cycle for configuration process so when RAM mapping is enabled it requires additional pipeline registers and some logic around it to access RAM correctly. Figure 24 below shows the whole design resources used when RAM mapping was enabled on a block using the structure described above. Red values in the figure indicate the increase of resources compared to the design that had RAM mapping disabled. The resources used without RAM mapping can be found from Figure 16.

Multipliers	6	+ 0
Adders/Subtractors	95	+ 28
Registers	484	+ 270
RAMs	1	+ 1
Multiplexers	228	+ 136

Figure 24. High level resource report when RAM mapping is enabled.

Figure 24 shows that now one RAM was generated but it also more than doubled the amount of registers and multiplexers, and also generated additional adders/subtractors. To be beneficial, this should have decreased the amount of registers utilized, however, it only made the HDL design worse.

In this work, the example design was not optimized by built-in optimization methods but the algorithms were written in a way to optimize timing and area for FPGA target. The optimization was done by following the good coding rules introduced in section 4.3.

The first version of the design had feed-through type blocks. Signals were assigned to the output at the end of the algorithm. This generated long data path delays over some the blocks because the signals were registered only in a few parts of the design. By creating registers into outputs of each block the data path delays decreased and none of the generated blocks were on critical path anymore. LUTs' delay became dominant on critical path. The critical path of the design is shown in Figure 17. The generated design did not need any further optimization to meet FPGA timing requirements.

7.2. Optimization for ASIC target

VHDL targeting ASIC has different requirements than one targeting FPGA. Timing requirements are stricter than for FPGA and path delays are critical. The original hand-written model of the example design was targeted for ASIC running at clock frequency of 491 MHz.

The FPGA targeted version of generated model was able to run at 128,5 MHz so further optimization was required for ASIC target. Optimization was started by running ASIC synthesis targeting for 491 MHz frequency to point out the critical paths. The synthesis tool used was Synopsys Design Compiler. The timing report of the original generated design is shown in Figure 25.

```
Timing Path Group 'IO_Clk'            Timing Path Group 'clk'
--------------------------------      --------------------------------
Levels of Logic:            3.00      Levels of Logic:            42.00
Critical Path Length:       0.37      Critical Path Length:        3.04
Critical Path Slack:       -0.87      Critical Path Slack:        -2.61
Critical Path Clk Period:   2.03      Critical Path Clk Period:    2.03
Total Negative Slack:      -9.79      Total Negative Slack:    -4742.87
No. of Violating Paths:    19.00      No. of Violating Paths:   3258.00
Worst Hold Violation:       0.00      Worst Hold Violation:        0.00
Total Hold Violation:       0.00      Total Hold Violation:        0.00
No. of Hold Violations:     0.00      No. of Hold Violations:      0.00
--------------------------------      --------------------------------
```

Figure 25. ASIC synthesis timing report of the original model.

As can be seen from Figure 25, in both timing path groups, there exists negative slack which means that some of the data paths are too slow and the design is not functional with the clock frequency. Either the clock frequency has to be decreased or data paths have to be shortened to make the design work on the desired clock frequency. Area of the generated model compared to the hand-written model is presented in Table 5 below.

Table 5. The original model area compared to the hand-written model

Area	Percentage of the hand-written model
Combinational	128,9 %
Sequential	51,6 %
Total	113,6 %

The timing report also points all the paths breaking the timing requirement. An example of this is presented in Appendix 5. From the report, it can be seen the data required time and the data arrival time, thus all the logic between the registers. In this case, data paths were to be shortened by adding register in combinational structures. This increases area but improves the maximum clock frequency.

Optimization was started from the longest data paths that were in blocks including multiplication, rounding and saturation. To shorten the data paths, pipelining had to be added between the logic as described in section 2.2.1. The pipelining in this case was done by dividing the blocks in smaller blocks with registered outputs and this

generated registers after each larger logical operation mentioned above. In the Simulink model, this was done by writing the MATLAB algorithms again in smaller serial blocks using persistent variables to generate registers in between of the blocks. Illustration of the scenario is shown in Figure 26 below.
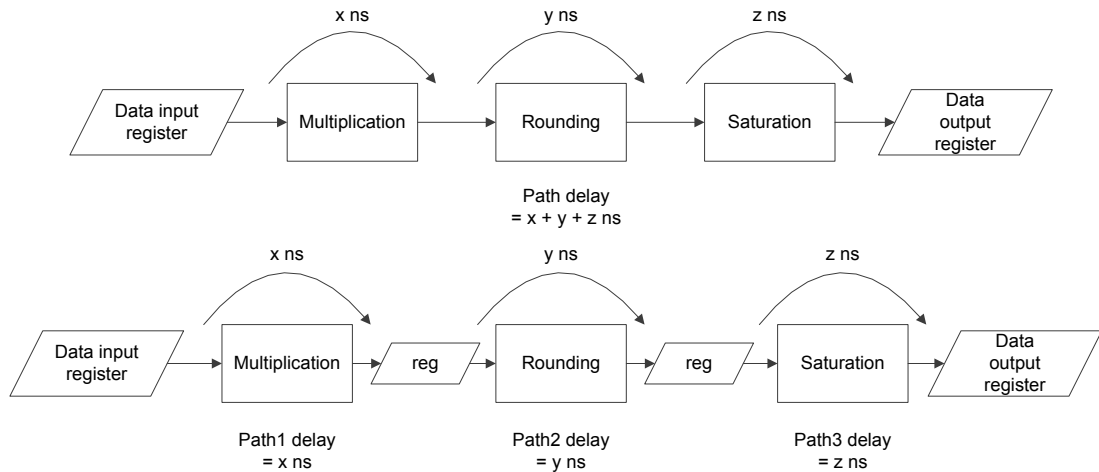
Figure 26. Illustration of adding registers in a long logic path.

Another structure causing long data paths was when indexed input variables were used in conditional statement. To shorten the data paths, the indexed variables were selected and registered before the usage in a functional block. All the other input signals to the original block were delayed by "Unit Delay"-blocks to generate registers and one clock cycle delay for synchronization. The variable selection was done with "Index Vector"-block. This is presented in Figure 27.

```
23        %Calculate the power deviation, format: 1.7.5 -> 1.8.5
24 -      if (start)
25 -          PtxCarDev_tmp(:) = Ptx - Pmax(channelID+1);
26 -          valid_reg = true;
27        end
```

```
23        %Calculate the power deviation, format: 1.7.5 -> 1.8.5
24 -      if (start)
25 -          PtxCarDev_tmp(:) = Ptx - Pmax;
26 -          valid_reg = true;
27        end
```
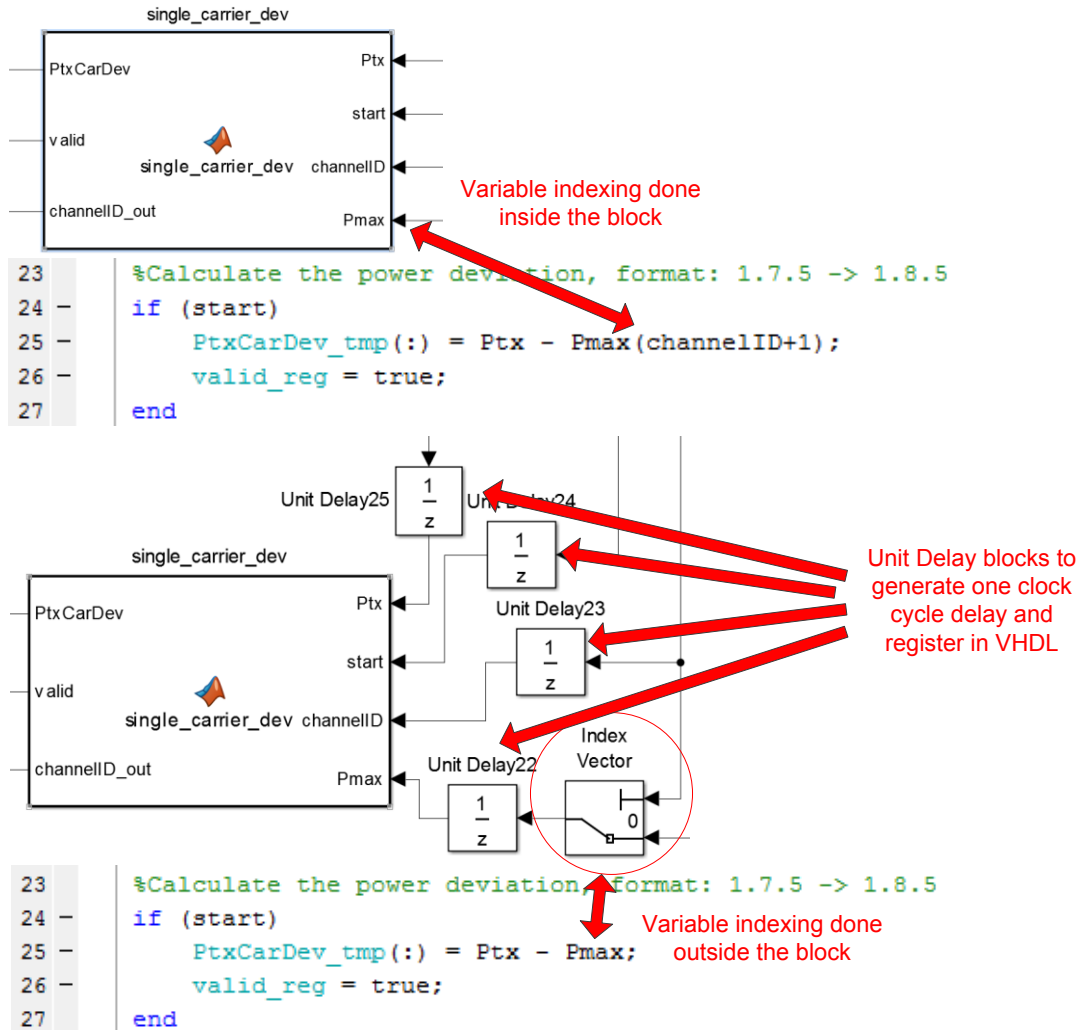
Figure 27. Illustration of performing variable indexing inside and outside of a functional block.

Using these two techniques on the design improved the ASIC synthesis results. The timing report with the more ASIC optimized model is presented in Figure 28.

Timing Path Group 'IO_Clk'		Timing Path Group 'clk'	
Levels of Logic:	0.00	Levels of Logic:	22.00
Critical Path Length:	0.38	Critical Path Length:	1.85
Critical Path Slack:	0.13	Critical Path Slack:	-0.47
Critical Path Clk Period:	2.03	Critical Path Clk Period:	2.03
Total Negative Slack:	0.00	Total Negative Slack:	-316.31
No. of Violating Paths:	0.00	No. of Violating Paths:	861.00
Worst Hold Violation:	0.00	Worst Hold Violation:	0.00
Total Hold Violation:	0.00	Total Hold Violation:	0.00
No. of Hold Violations:	0.00	No. of Hold Violations:	0.00

Figure 28. ASIC synthesis timing report of the optimized design.

From Figure 28, it can be seen that all the signals in timing path group "IO_Clk" are meeting the requirements. In timing path group "clk", the critical path slack is roughly one fifth of the original slack and also the total negative slack decreased into around one twentieth of the original. However, the model is not meeting the timing requirements and further optimization would be required. Due to lack of time, no further optimization was done but it seems like the flow is capable of generating ASIC level HDL code in speed wise.

During the ASIC optimization, also some combinatorial logic structures were optimized by forcing signals in blocks to certain data types and bit widths. In some cases, if the signals are not clearly declared in MATLAB function or System Object HDL Coder may generate unnecessary multiplexing and rounding structures. Example of this is presented in Appendix 6. ASIC optimized design provided better area report than FPGA optimized design. The area of the ASIC optimized design compared to the hand-written model is presented in Table 6.

Using Simulink library components saves from creating unnecessary structures in the HDL. The components are resource optimized for HW generation and should be used to model all the parts of the design that can be trivially made. If the design includes arithmetic operations or other structures that cannot be trivially built by the library components, the user can write the algorithms in MATLAB functions and System Objects. This kind of hybrid flow also generates synthesizable VHDL.

Table 6. The ASIC optimized model area compared to the hand-written model

Area	Percentage of the hand-written model
Combinational	68,5 %
Sequential	45,5 %
Total	70,7 %

Table 6 presents that optimization improved the area results in both combinational and non-combinational area. Increasing pipelining should have generated more non-combinational resource but together with optimized combinational structures it actually removed some of the earlier unnecessary registers from the design.

The results prove that the flow is well suited to produce not only prototyping HDL for FPGA target but also fast and area effective HDL for ASIC target. However, for ASIC target, the model has to be written in Simulink or MATLAB very similarly as it would be written in RTL. Using the flow for ASIC requires RTL knowledge from the designer.

8. DISCUSSION

In this chapter, MathWorks HLS flow for rapid prototyping is discussed and compared to general FPGA flow introduced in section 3.2. Design flow, verification flow, design quality and future development are evaluated.

8.1. Performance and time usage from algorithm to FPGA prototype

HDL Coder generated VHDL surprises with it is performance shown in Chapter 6. The generated code has good FPGA synthesis results by utilizing fewer resources than the hand-written model and also being able to run on higher clock frequency on FPGA. Moreover, generated code is human readable and includes comments from MATLAB algorithm and traceability backwards to MATLAB/Simulink model through links.

Below, is presented the time usage of each phase of the design cycle. The original algorithm had to be completely rewritten to produce reasonable RTL. Moreover, the work was done without previous work experience of algorithm coding, RTL coding, logic synthesis or FPGA prototyping. Verification was done only by co-simulation and FIL described in sections 5.1 and 6.2, no other time consuming verification methods were used. All FPGA technology, files were provided so there was no need to manually setup new FPGA for logic synthesis. These should be taken into account when analyzing design flow times. Relative design flow times are presented in Table 7.

Table 7. Time usage from the example algorithm into FPGA prototype

Phase	Algorithm	Verification	Logic Synthesis (FPGA)	FPGA verification	Total
Time(weeks)	6 (60 %)	1 (10 %)	1 (10 %)	2 (20 %)	10 (100 %)

Generated VHDL code quality seems feasible for prototyping. If HDL Coder is able to produce similar quality HDL as in the example case for other designs, it reduces design flow times, therefore, improving the SoC development flow. Code generation also reduces RTL verification times because in ideal case generated code is bit-accurate, cycle-accurate and flawless design that can be synthesized into FPGA model. RTL verification methods and 3rd party tools support improve the verification times slightly compared to the manual flow. On the other hand, verification work on the model is increased, but iteration speed from the algorithm to RTL is improved. In Figure 29 below, is shown an illustration how the HLS flow could improve the FPGA prototyping flow and in Figure 30 is illustrated the effect on the whole ASIC design flow time.
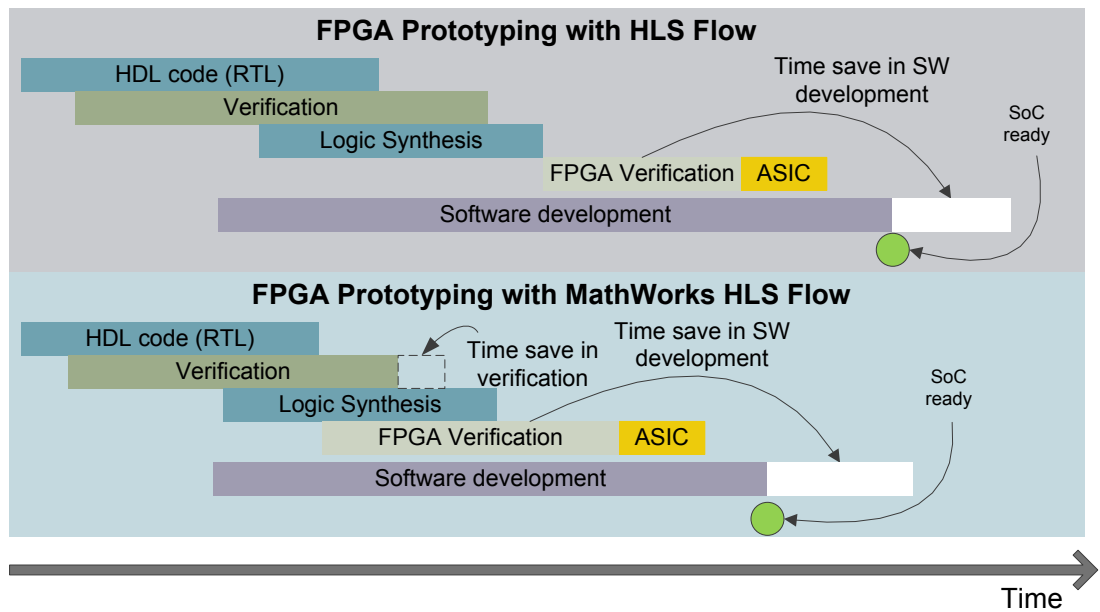
Figure 29. Possible effect of MathWorks HLS flow for FPGA prototyping flow time.
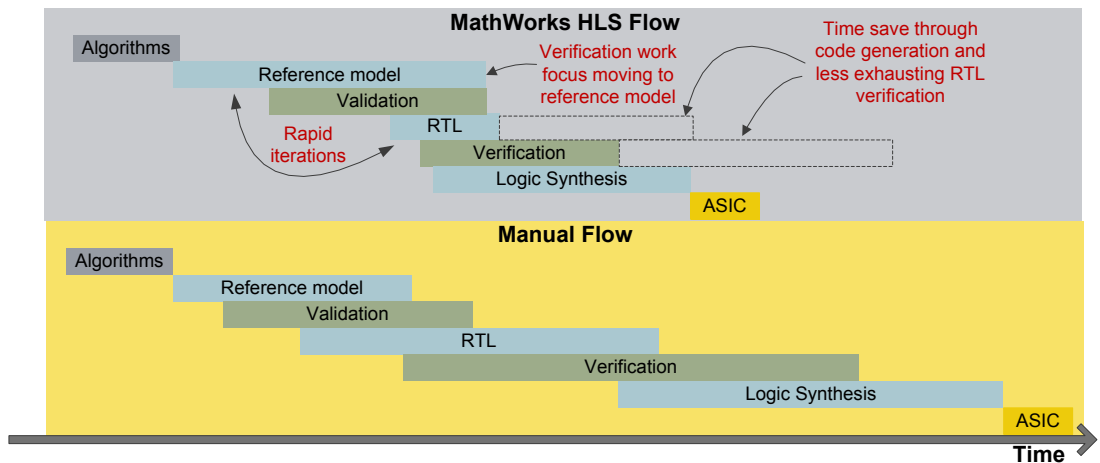
Figure 30. Possible effect of using MathWorks HLS flow for ASIC design flow time.

From the figures above, it can be said that using the workflow may have some actual benefit on design flow times and may be feasible for FPGA prototyping. The actual ratio of benefit is hard to derive since the flow was tested only with one example block and functionality or effectiveness with all type of algorithms cannot be guaranteed. To verify this, requires using the workflow with more thorough, actual large prototyping case.

8.2.   Code generation targeting production quality

Production quality means HDL that is such a quality that can be used for production and needs no further optimization. HDL Coder can produce production quality RTL at least when creating common sequential and combinational logic structures. This means that HDL Coder can be used to create some parts or the whole IP targeting for production.

   To achieve this requires knowing good algorithm coding rules in MATLAB and also understanding what structures are good in RTL. Table 8 below shows some methods to fine tune algorithm to produce high quality RTL.

Table 8. Fine tuning methods for targeting production quality RTL

Production Quality Target	Method
Minimizing critical path delay to maximize operating frequency	Adding registers to output ports of design blocks or into long logic paths. This can be done either by adding them trough persistent variables in the algorithm or using HDL Coder optimization methods.
Loosen delay on unnecessarily fast data path for area optimization	Removing registers from algorithm on data paths that meet timing requirements without working on high frequency.
Minimizing signal bit widths to improve performance and area optimization	Defining signal bit widths to precisely cover signal range in fixed-point data types in algorithm or Fixed-Point Converter.
Minimize clock enables for area optimization	By default, HDL Coder maps registers with clock enable, enable "Minimize clock enables" feature to reduce the amount of clock enable logic if design contains registers without clock enables (Cannot be used together with resource sharing, RAM mapping or loop streaming).
Use RAM mapping for larger registered variables for area optimization	Set "RAM mapping threshold" to a value that registered variables with greater bit width are mapped to RAM rather than to registers.
Use monotonically increasing loop counters for area optimization	Set loop counter increments to 1, increments other than 1 can require additional adders in hardware.
Maximize performance by utilizing Simulink library blocks	The library components are optimized for hardware target and are less risky to use compared to MATLAB functions or System Objects.

   By following these methods, it is possible to achieve good quality in RTL that can be used if not completely at least partly for production. This feature shows future potential for automating HDL generation straight from the algorithm without any manual work. When the HLS tools are mature enough to reliably produce production quality HDL, it will have a great impact on design flow times by speeding up the design cycle in a half or more of the current cycle.

8.3. Future development

In this section, future development to improve the flow for SoC prototyping is discussed. Integration with comprehensive FPGA environment and IP packaging with generic interface are covered.

8.3.1. Future view with high-performance FPGA environment

MathWorks HLS workflow does not support all prototyping environments by default but it has an API to connect new FPGA boards by user. Support for high-performance FPGA environments is required to enable better performance for ASIC prototyping and especially for large scale SoC system level prototyping.

FPGA prototyping environments differ from general FPGA boards by including complete toolset and versatile connectivity. They generally utilize a powerful FPGA and provide a lot of memory and I/O resources, and a design might not require partitioning in smaller pieces. Therefore, they suit well for SoC level prototyping. One example of high-performance environment is Synopsys HAPS, which utilizes Xilinx Virtex-7 FPGA. As an example, HAPS's benefits compared to general FPGA boards are shown in Figure 31.
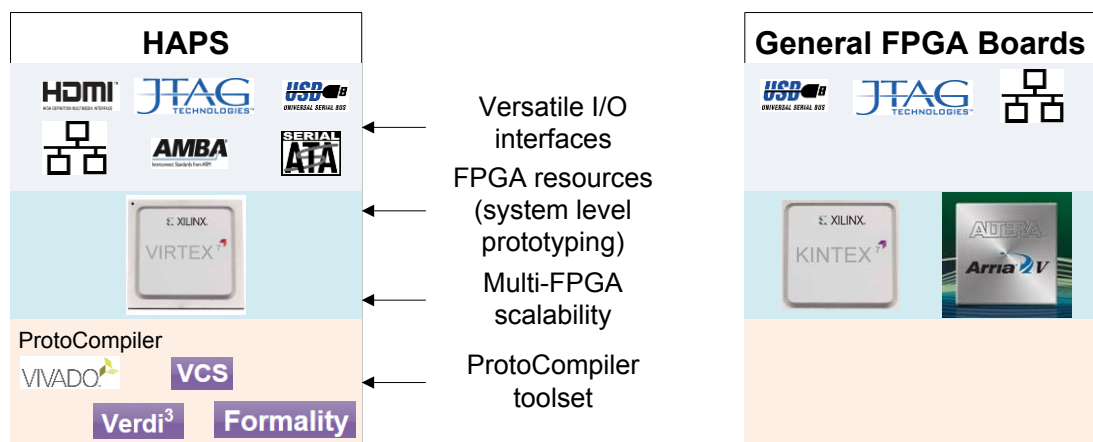
Figure 31. HAPS prototyping environment compared to general FPGA boards.

HAPS prototyping environment consists of HAPS system, ProtoCompiler software, host PC and peripherals. The prototyping environment provides high-performance tools and connectivity to improve ASIC prototyping. [20]

Integration of a high-performance FPGA environment into MathWorks HLS flow improves its performance for SoC prototyping flow by increasing the resources that can be used for the design: more memory and I/Os, higher operating frequency and better connectivity. Not only the IP testing will improve, but this could make it possible to do system level prototype verification.

8.3.2. *IP packaging and RTL verification with existing test bench configuration*

IP packaging with common interface is one key thing for SoC prototyping. It provides connectivity between all designed IPs and makes the blocks easy to implement in a design. Generic variables in IPs HDL code provide scalability for the block depending on the design.

The example design was meant to be packaged with Advanced eXtensible Interface (AXI) interface to provide a possibility to implement the generated block in existing design. Together with generic variables, this would have provided the possibility to test the generated model in RTL simulation with the existing test bench in actual design. AXI interface packaging and generic variables generation were left out of the scope due to lack of time.

This should be implemented and tested in the future to verify that common interfaces can be generated and are functional. Illustration of the IP packaging with generic interface and RTL verification configuration is shown in Figure 32.
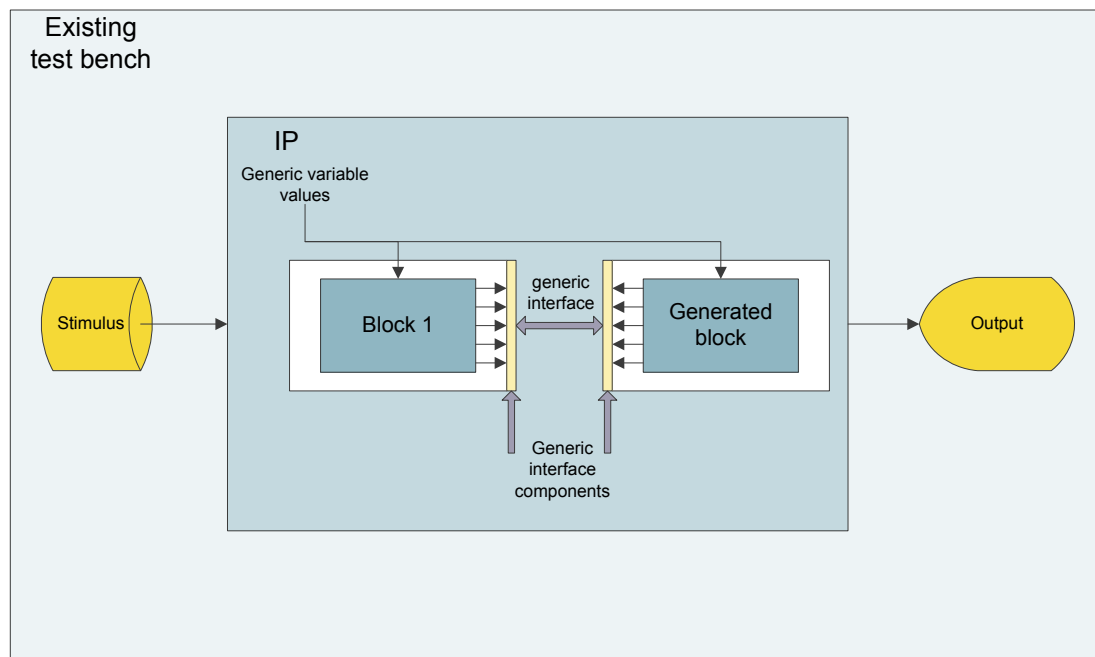

Figure 32. IP packaging example with generic interface components in RTL simulation configuration.

9. CONCLUSION

Increasing wireless data usage sets stricter requirements for SoCs targeted for telecommunication systems. Growing complexity of ASIC designs increases manual algorithm and RTL coding which further makes verification more laborious. HLS tools are starting to be a feasible alternative to be used to generate RTL prototype from algorithm model. This truncates time from algorithm to FPGA prototype, therefore is suitable for rapid prototyping.

The aim of this thesis was to study how well MathWorks HLS workflow suits for rapid prototyping. The flow was studied with an example IP block that scales and limits the power of IQ-data in telecommunication SoC. The goal was to examine the speed of the entire flow, good coding rules to generate synthesizable VHDL, resource utilization on FPGA and ASIC, design speed and possible production code quality.

HDL Coder was able to generate human readable production quality VHDL code when algorithm was written by following good coding rules optimized for hardware. Bad quality RTL was generated if the algorithm was written from perspective of simulation speed and large data vectors were processed at once inside a function. Generated RTL was verified in co-simulation and validation model simulation without any errors.

Logic synthesis was done on the generated RTL and it provided promising results. The generated model utilized fewer resources than the original and it was able to run on higher clock frequency on FPGA. The original model was targeted on ASIC so the results are not perfectly comparable. ASIC synthesis was done on the generated model but it didn't meet the timing requirements. By optimizing the algorithm better results were reached. The results show that the HLS flow can provide good quality design when executed by implementing best practices targeting for HW.

Finally, the design was verified on Altera's FPGA board in FPGA-in-the-loop configuration. The design was successfully programmed on the FPGA through JTAG-connection. FPGA-in-the-loop simulation was used to verify the functionality of the FPGA design with Simulink stimulus. The output from the FPGA was matching the algorithm model and no errors were discovered.

RTL verification takes roughly 70% of the whole design cycle. The HLS flow can be used to improve the prototyping flow by automating the RTL generation and also decreasing the RTL verification times by moving the focus on the algorithm verification. Early prototype also enables earlier SW development which further improves the design flow.

Full integration into SoC prototyping flow requires connection to a powerful FPGA prototyping environment e.g. Synopsys HAPS. However, it is already useful for prototyping small IPs on alternative FPGA configurations. Furthermore, tight co-operation with algorithm and RTL designers is required to implement this kind of flow efficiently.

In general, rapid prototyping with HLS tools seems to be the future way to correspond the increasing workload on exhausting prototyping phases of complex ASIC designs. It might be even possible to generate the production code or part of it with these tools in the future.

10. REFERENCES

[1]   Mollick E., (2006) Annals of the History of Computing, IEEE (Volume: 28, Issue: 3), Establishing Moore's Law. IEEE, 14 p., 62-75.

[2]   Harte L. & Bowler D. (2004) Introduction to Mobile Telephone Systems: 1G, 2G, 2.5G, and 3G Wireless technologies and Services. Althos Publishing, 43 p.

[3]   Fingeroff (2010) High-Level Synthesis Blue Book. Xlibris, 272 p.

[4]   Black D., Donovan J., Bunton B. & Keist A. (2010) SystemC: From the Ground Up, Second Edition. Springer, New York, USA, 361 p.

[5]   Coussy P. & Moraviec A. (2008) High-Level Synthesis: From Algorithm to Digital Circuit. Springer Science + Business Media B.V., Dordrecht, Netherlands, 297 p.

[6]   Kaeslin H. (2008) Digital Integrated Circuit Desing From VLSI Architectures to CMOS Fabrication. Cambridge University Press, Cambridge, UK, 845 p.

[7]   Horváth P. (2014) RTL Optimization Techniques. (Accessed 24.2.2015) URL: http://www.eet.bme.hu/~horvathp/contents/aramkortervezes/eloadasok/07_RTL_Optimization_Techniques.pdf

[8]   Qing W., Pedram M. & Xumwei W. (2000) Clock-Gating and Its Application to Low Power Design of Sequential Circuits. Circuits and Systems I: Fundamental Theory and Applications. IEEE, 6 p., 415-420.

[9]   Qi W. & Sumit R. (2003) RTL Power Optimization with Gate-level Accuracy. International Conference of Computer Aided Design. IEEE, 7 p., 39-45.

[10]   Quora (2015) Semiconductors: What are the differences between static and dynamic power consumption in CMOS integrated circuit? (Accessed 11.6.2015) URL: http://www.quora.com/Semiconductors/What-are-the-differences-between-static-and-dynamic-power-consumption-in-CMOS-integrated-circuits

[11]   Kim N.S., Austin T., Baauw D., Mudge T., Flautner K., Hu J.S., Irwin M.J., Kandemir M., Narayanan V. (2003) Computer (Volume: 46, Issue: 12) Leakage Current: Moore's Law Meets Static Power. IEEE, p. 68-75.

[12]   Rashinkar P, Paterson P. & Singh L. (2001) System-on-a-chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 372 p.

[13]   Donald D. & Moorby P. (2002) The Verilog Hardware Description Language. Springer Science + Business Media, Inc., New York, USA, p. 35-71.

[14]   National Instruments (2012) Introduction to FPGA technology: Top 5 Benefits (Accessed 27.2.2015) URL: http://www.ni.com/white-paper/6984/en/

[15]   Xilinx (2015) What is a FPGA? (Accessed 27.2.2015) URL: http://www.xilinx.com/fpga/

[16]   Altera (2015) FPGAs (Accessed 27.2.2015) URL: http://www.altera.com/products/fpga.html

[17]   Amos D., Lesea A. & Richter R. (2011) FPGA-Based Prototyping Methodology Manual. Synopsys, Inc., Mountain View, CA, USA, 470 p.

[18]   Electronic Engineering Journal (2014) Xilinx vs. Altera (Accessed 27.2.2015) URL: http://www.eejournal.com/archives/articles/20140225-rivalry/

[19]   Synopsys (2015) FPGA Based Prototypes Ideal for Prototyping ASIC and IP (Accessed 2.3.2015) URL: http://www.synopsys.com/Prototyping/FPGABasedPrototyping/haps-dx/Pages/default.aspx

[20]   Synopsys (2015) HAPS Datasheets (Accessed 3.3.2015) URL: http://www.synopsys.com/Prototyping/FPGABasedPrototyping/pages/Datasheets.aspx

[21]   Xilinx (2015) FPGA vs. ASIC (Accessed 27.2.2015) URL: http://www.xilinx.com/fpga/asic.htm

[22]   Churiwala S. & Garg S. (2011) Principles of VLSI RTL Design. Springer Science+Business Media, New York, USA, 182 p.

[23]   Cofer R.C. & Harding B. (2006) Rapid System Prototyping with FPGAs. Elsevier Inc., Burlington, USA, 247 p.

[24]   MathWorks (2015) What Are System Objects? (Accessed 15.4.2015) URL: http://se.mathworks.com/help/comm/gs/what-are-system-objects.html

[25]   MathWorks (2015) HDL Coder User's Guide (Accessed 20.4.2015) URL: http://cn.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf

[26]   MathWorks (2015) How Acceleration Modes Work (Accessed 26.5.2015) URL: http://se.mathworks.com/help/simulink/ug/how-the-acceleration-modes-work.html

[27]   MathWorks (2015) HDL Verifier User's Guide (Accessed 26.5.2015) URL: http://cn.mathworks.com/help/pdf_doc/hdlverifier/hdlv_ug_book.pdf

11. APPENDICES

Appendix 1   An example of parallel and serial structure of MATLAB algorithm

Parallel structure with variable indexing that generates three multipliers

```matlab
1  function output = simple_multiplication(datax, datay, start)
2  %simple multiplication function that multiplies 1st, 2nd and 3rd values of
3  %vectors datax and datay, and sets those to output vector
4
5  %datax = [1 2 3] and datay = [4 5 6] are vectors
6
7  %initialize output to a vector of zeros
8  output = [0 0 0];
9
10 %multiply values when start flag is set "true"
11 if (start)
12     output(1) = datax(1) * datay(1);
13     output(2) = datax(2) * datay(2);
14     output(3) = datax(3) * datax(3);
15 end
```

Serial structure with variable indexing that generates one multiplier and two 3-to-1 multiplexers

```matlab
1  function output = simple_multiplication(datax, datay, start)
2  %simple multiplication function that multiplies 1st, 2nd and 3rd values of
3  %vectors datax and datay, and sets those to output vector
4
5  %datax = [1 2 3] and datay = [4 5 6] are vectors
6
7  %initialize output to a vector of zeros
8  output = [0 0 0];
9
10 %multiply values when start flag is set "true"
11 if (start)
12     for i = 1:3
13         output(i) = datax(i) * datay(i);
14     end
15 end
```

Appendix 2   Downlink data scaling and power limitation simulation configuration
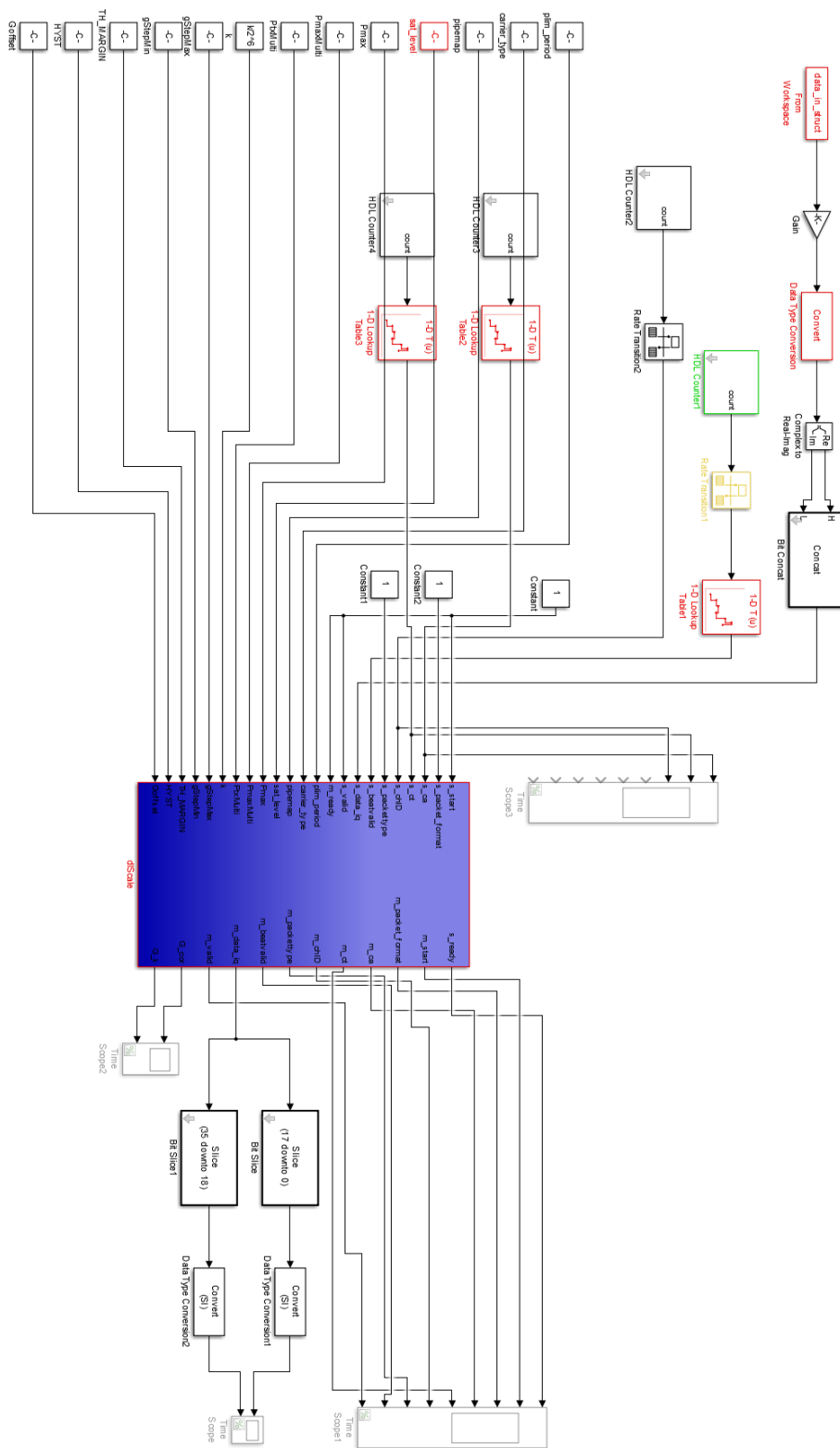
Appendix 3   MATLAB algorithm synthesis to VHDL example

MATLAB

```
24   %add and saturate, format: 1.5.5 + 1.5.5 -> 1.6.5 -> 1.5.5
25 — if (start)
26 —     tmp_add = fi(((G_cor + G_offset(channelID+1)), 1, 11, 5, fm);
27 —     tmp_sat = tmp_add;
28 —     valid_reg = true;
29   end
```

MATLAB comments

Link to MATLAB design ◆

VHDL

```
154   --add and saturate, format: 1.5.5 + 1.5.5 -> 1.6.5 -> 1.5.5
155   IF start = '1' THEN
156   --'<S24>:1:26'
157   --'<S24>:1:25'
158   add_temp := resize(channelID unsigned, 13) + to_unsigned(16#0001#, 13);
159   sub_cast := signed(resize(add_temp, 32));
160   add_temp_0 := (resize(G_cor_signed, 13)) + (resize(G_offset(to_integer(sub_cast - 1)), 13));
161   tmp_sat_next <= add_temp_0(10 DOWNTO 0);
162   --'<S24>:1:27'
163   --'<S24>:1:28'
164   valid_reg_next <= '1';
165   END IF;
```

Line number in MATLAB

◆ ◆   ◆ ◆

Appendix 4   Nested conditional statement on persistent variables in "for"-loop

```matlab
for j = 0:31
%Check the power measurement sample count for each case
%setting the value in fixed-point data type, format: 0.18.0
if (txpipe == 0)
    switch carrier_type(j+1)
        case 0
            cmrSamples(j+1) = plim_period_x16(1);
        case 1
            cmrSamples(j+1) = plim_period_x12(1);
        case 2
            cmrSamples(j+1) = plim_period_x8(1);
        case 3
            cmrSamples(j+1) = plim_period_x4(1);
        case 4
            cmrSamples(j+1) = plim_period_x2(1);
        case 5
            cmrSamples(j+1) = plim_period_x1(1);
        case 6
            cmrSamples(j+1) = plim_period_x2(1);
        case 7
            cmrSamples(j+1) = plim_period_x4(1);
        case 8
            cmrSamples(j+1) = plim_period_x6(1);
        case 9
            cmrSamples(j+1) = plim_period_x12(1);
        case 10
            cmrSamples(j+1) = plim_period_x12(1);
        case 15
            cmrSamples(j+1) = plim_period_gsm(1);
        otherwise
            cmrSamples(j+1) = cast(zeros, 'like', cmrSamples);
    end
elseif (txpipe == 1)
    switch carrier_type(j+1)
        case 0
```

Appendix 5   An example timing report displaying data paths with negative slack

```
Startpoint: u_data_branching/data_I_reg_reg[11]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: u_multiplier/round_I_reg[13]
            (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model        Library
------------------------------------------------------
dlScale            lsi28_rc_wlm_1.0       lsi28_rc_wlm

Point                                                   Incr       Path
------------------------------------------------------------------------
clock clk (rise edge)                                   0.00       0.00
clock network delay (ideal)                             0.00       0.00
u_data_branching/data_I_reg_reg[11]/CP (SDFPRQX4MV0SI35D)
                                                        0.00 #     0.00 r
u_data_branching/data_I_reg_reg[11]/Q (SDFPRQX4MV0SI35D)
                                                        0.15       0.15 f
u_multiplier/U699/Z (BUFX6BV0SI35D)                     0.07       0.22 f
u_multiplier/U2070/Z (XNOR2X4MV0SI35D)                  0.06       0.29 r
u_multiplier/U1917/Z (NAND2X6MV0SI35D)                  0.05       0.34 f
u_multiplier/U3120/Z (BUFX8BV0SI35D)                    0.07       0.41 f
u_multiplier/U2219/Z (OAI22X4MV0SI35D)                  0.06       0.47 r
u_multiplier/U3214/CO (ADDFX2MV0SI35D)                  0.13       0.60 r
u_multiplier/U1024/S (ADDFX1MV0SI35D)                   0.18       0.78 f
u_multiplier/U2164/S (ADDFX2MV0SI35D)                   0.21       0.99 r
u_multiplier/U2406/S (ADDFX2MV0SI35D)                   0.20       1.19 r
u_multiplier/U1966/Z (NOR2X6MV0SI35D)                   0.06       1.24 f
u_multiplier/U1084/Z (INVX6BV0SI35D)                    0.04       1.28 r
u_multiplier/U1621/Z (NAND3X6MV0SI35D)                  0.04       1.32 f
u_multiplier/U811/Z (NOR2X6MV0SI35D)                    0.04       1.36 r
u_multiplier/U1541/Z (OAI21X6MV0SI35D)                  0.05       1.40 f
u_multiplier/U1501/Z (NOR2X6MV0SI35D)                   0.04       1.45 r
u_multiplier/U3396/Z (OAI21X8MV0SI35D)                  0.04       1.49 f
u_multiplier/U3397/Z (NAND2X8MV0SI35D)                  0.05       1.53 r
u_multiplier/U3403/Z (AOI21X8MV0SI35D)                  0.07       1.60 f
u_multiplier/U2048/Z (BUFX8BV0SI35D)                    0.08       1.68 f
u_multiplier/U694/Z (OAI21X6MV0SI35D)                   0.05       1.73 r
u_multiplier/U734/Z (XNOR2X4MV0SI35D)                   0.07       1.81 r
u_multiplier/U3464/Z (OAI21CNX4MV0SI35D)                0.04       1.85 f
u_multiplier/round_I_reg[13]/D (SDFPRQNX1MV0SI35D)      0.00       1.85 f
data arrival time                                                  1.85

clock clk (rise edge)                                   2.03       2.03
clock network delay (ideal)                             0.00       2.03
clock uncertainty                                      -0.50       1.53
u_multiplier/round_I_reg[13]/CP (SDFPRQNX1MV0SI35D)     0.00       1.53 r
library setup time                                     -0.15       1.38
data required time                                                 1.38
------------------------------------------------------------------------
data required time                                                 1.38
data arrival time                                                 -1.85
------------------------------------------------------------------------
slack (VIOLATED)                                                  -0.47
```

Appendix 6   Example of a structure creating unnecessary logic in a design

```
if (run)
    mk_tmp = Gk;                    ← Data type not
    %Multiplied branches, format: 1.0.15 * 0.4.14 -> 1.4.29 (output format is set automati
    mul_I = data_I*mk_tmp;
    mul_Q = data_Q*mk_tmp;
```

↓

```
259    IF run = '1' THEN
260        --'<S37>:1:41'
261        --Multiplied branches, format: 1.0.15 * 0.4.14 -> 1.4.29 (output format is set automatically by m
262        --'<S37>:1:44'
263        mul_temp := data_I_signed * signed(resize(Gk_unsigned, 19));
264        IF (mul_temp(34) = '0') AND (mul_temp(33) /= '0') THEN
265          mul_I := "01111111111111111111111111111111111";
266        ELSIF (mul_temp(34) = '1') AND (mul_temp(33) /= '1') THEN
267          mul_I := "10000000000000000000000000000000000";        ← Two multipliers and
268        ELSE                                                        two multiplexers
269          mul_I := mul_temp(33 DOWNTO 0);
270        END IF;
271        --'<S37>:1:45'
272        mul_temp_0 := data_Q_signed * signed(resize(Gk_unsigned, 19));
273        IF (mul_temp_0(34) = '0') AND (mul_temp_0(33) /= '0') THEN
274          mul_Q := "01111111111111111111111111111111111";
275        ELSIF (mul_temp_0(34) = '1') AND (mul_temp_0(33) /= '1') THEN
276          mul_Q := "10000000000000000000000000000000000";
277        ELSE
278          mul_Q := mul_temp_0(33 DOWNTO 0);
279        END IF;
```

```
41    if (run)
42        mk_tmp = fi(Gk, 1, 19, 14);        ← Data type
43        %Multiplied branches, format: 1.0.15 * 0.4.14 -> 1.4.29 (output format is set automatic    defined
44        mul_I = data_I*mk_tmp;
45        mul_Q = data_Q*mk_tmp;
```

↓

```
258    IF run = '1' THEN
259        --'<S37>:1:41'
260        --'<S37>:1:42'
261        mk_tmp := signed(resize(Gk_unsigned, 19));
262        --Multiplied branches, format: 1.0.15 * 0.4.14 -> 1.4.29 (output format is set automatically by matlab)
263        --'<S37>:1:44'
264        mul_I := data_I_signed * mk_tmp;
265        --'<S37>:1:45'        ← Two multipliers
266        mul_Q := data_Q_signed * mk_tmp;
```