NTNU

Norwegian University of
Science and Technology

High-Level Synthesis for Application-Specific Integrated Circuit Implementation using LegUp

Jørgen F Holmefjord

Master of Science in Electronics
Submission date: June 2016
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Title:	High-Level Synthesis for Application-Specific Integrated Circuit Implementation using LegUp
Student:	Jørgen Frydenlund Holmefjord

Problem description:

Architectural exploration is a long and complex process where a number of hardware architectures are built and evaluated based on minimum performance requirements and worst-case operational scenarios. With this method, satisfactory results can be achieved if a diverse number of candidates are produced. However, the number of architectures to be evaluated is limited by time and engineering resources. In this context, High Level Synthesis (HLS) is a compelling alternative to shorten the development time, and consequently, increasing the number of architectures that can be evaluated during the exploration. Furthermore, by automating the entire architecture exploration process, the optimization engine can take advantage of the higher level of abstraction and generate far more and diverse architectures than it would be possible by parametrized RTL.

During the autumn of 2015, a project was conducted to evaluate the open-source HLS tool LegUp [13], and whether it can be used in a framework for architectural exploration of digital hardware. During the work with the project some fundamental issues were exposed, limiting the tool's usefulness for our initial intentions. The main issues are related to input and output of the generated modules, structure of memory management, and size of signals.

The goal of this master thesis is to resolve the encountered issues, and if time allows it, start building an initial framework for architectural exploration.

Possible sub-tasks and goals of this thesis are:

- Explore the two approaches proposed in the project for resolving the encountered issues.
- Determine if LegUp's C-like memory-bound architecture can be eliminated by de-referencing pointers or turn memory elements into generic signals.
- Re-evaluate if LegUp is capable of generating synthesizable Verilog HDL for ASIC implementation and if it can be used in a framework for automatic architectural exploration.

- Set an initial HLS framework for architectural exploration of digital hardware.
- Create scripts to automate simulation, synthesis, and power dissipation extraction.
- Integrate Nordic Semiconductor's coding style and practices into LegUp Verilog libraries, i.e. interfaces, parameters, naming conventions, power/clock domains, etc.

Responsible professor: Kjetil Svarstad, IET
Supervisor: Isael Diaz, Nordic Semiconductor

Abstract

Low power and small area are becoming increasingly important and highly demanded in large System-on-Chip (SoC) designs, incorporating billions of transistors. This entails that the typical design methodology is no longer sufficient, if hardware manufacturers want to supply the best product on the market. Architectural exploration is an important part of the design process, where multiple designs are built and evaluated in terms of area, performance, and power consumption. High-level synthesis (HLS) is a compelling alternative to reduce the effort put into architectural exploration. By using HLS in a framework for architectural exploration of digital hardware, the number and diversity of architectural variations that can be generated and evaluated is far greater than what could have been done manually.

During a previous project, the HLS-tool *LegUp* was explored. The goal was to see if the tool could be used the described framework. The conclusions from the project was that LegUp had some issues, limiting its ability to generate Register-Transfer Level (RTL)-code suitable for Application-Specific Integrated Circuit (ASIC) implementation.

This thesis presents a solution for an architectural exploration framework built on an adapted version of LegUp. The framework can generate a large amount of architectural variations of a design written in C, and run simulation, synthesis, layout and power analysis on each design. Randomized constraints are used in the framework to vary the output from the HLS-tool. The framework generate reports of area usage, maximum performance, and estimated power consumption for each of the generated designs, for the designer to be able to choose the best design based on trade-offs from the design specifications.

A *proof of concept* was conducted, running a FIR-filter design through the created framework. The result showed that a decrease in area of 13.28% and a decrease in power consumption of 9.52% could be achieved by selecting the best-case design over the worst-case design. These results indicate that the concept works. The overhead of the generated designs vary between 30-200%, making it impractical for hardware design. However, it looks like the fidelity of the results are high, making it possible to use the framework-results for selecting the best architecture. During the process of adapting LegUp to work with a tool-flow for ASIC implementations, some of the functionality of the tool have been lost. Some bugs has also been introduced and discovered. Before using the

created framework for any commercial purpose, these problems must be eliminated.

Sammendrag

Lavt effektforbruk og lite areal er stadig mer etterspurt i store design av enbrikkesystemer, bestående av milliarder av transistorer. Dette fører til at den typiske design-metoden ikke lenger er brukende, dersom maskinvare-produsentene ønsker å tilby det beste produktet på markedet.

Arkitektur-utforsking er en viktig del av designprosessen, hvor flere design skapes og blir evaluert i form av areal, ytelse, og effektforbruk. Høynivå syntese (HLS) er et attraktivt konsept for å redusere den samlede innsatsen designeren må legge ned i arkitektur-utforskingen. Ved å benytte HLS i et rammeverk for arkitektur-utforsking av digital maskinvare kan langt flere og mer varierte arkitekturelle variasjoner genereres og evalueres, sammenlignet med å utføre arbeidet manuelt.

I et tidligere prosjekt ble HLS-verktøyet *LegUp* utforsket. Målet var å undersøke om verktøyet kunne brukes i det beskrevne rammeverket. Konklusjonen fra prosjektet var at noen problemer med LegUp begrenser muligheten til å generere Register-Transfer Level (RTL)-kode egnet til implementering på applikasjonsspesifikk integrert krets (ASIC) arkitekturer.

Denne avhandlingen presenterer en løsning for et rammeverk for arkitek-tur-utforskring bygget på en tilpasset versjon av LegUp. Rammeverket kan generere et stort antall arkitekturelle variasjoner av et design skrevet i C, og kjøre simulering, syntese, layout, og effekt-analyse på hvert design. Randomiserte føringer benyttes i rammeverket for å generere varierte design fra HLS-verktøyet. Rammeverket genererer rapporter som beskriver arealbruk, maksimal ytelse, og beregnet effektforbruk for hvert design, slik at designeren kan velge det designet som passer best, basert på avveininger mellom viktige parametre fra designspesifikasjonen.

Et *konseptbevis* ble utført ved å kjøre et FIR-filter design gjennom rammeverket. Resultatet viste at en besparelse i areal på 13.28% og en besparelse i effektforbruk på 9.52% kan oppnås ved å velge det designet med best resultater over designet med dårligst resultater. Disse resultatene viser at konseptet fungerer. HLS-verktøyet genererer en økning i areal og effektforbruk sammenlignet med et tilsvarende design skrevet direkte i RTL-kode på mellom 30-200%, noe som gjør det lite økonomisk å benytte verktøyet til design av maskinvare. Forholdet mellom de genererte resultatene ser likevel ut til å stemme (høy *fidelity*), noe som gjør at rammeverk-resultatene kan benyttes til å velge arkitektur for designet.

Gjennom prosessen med å tilpasse LegUp til å generer kode som støttes av verktøyene i rammeverket har noe av den originale funksjonaliteten gått tapt. Noen feil har også oppstått og blitt oppdaget. Før rammeverket brukes til noen form for kommersielle formål må alle problemer som er beskrevet i denne rapporten elimineres.

Preface

This report is the result of the Master's thesis conducted during the spring of 2016, concluding a Master of Science degree in Electronics, Design of Digital Systems. The report is submitted to the Department of Electronics and Telecommunications at the Norwegian University of Science and Technology.

This work is a continuation of a specialization project conducted during the autumn of 2015. The project was proposed by Nordic Semiconductor in August 2015, and the continuation into a Master's thesis was a natural choice in January 2016. During the work with this thesis, I have learned a lot about the concept of high-level synthesis and how to implement integrated circuits, all the way from the planning stage until final layout.

I would like to thank my supervisors Isael Diaz at Nordic Semiconductor and professor Kjetil Svarstad from NTNU, for their guidance, support and feedback through this project. Finally, I want to thank my family and friends for their support, encouragement and inspirational discussions during this work and through the whole degree.

Trondheim, 2016-06-10

Jørgen Frydenlund Holmefjord

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ANSI American National Standards Institute.

ASIC Application-Specific Integrated Circuit.

CPU Central Processing Unit.

CSV Comma-Separated Values.

DDVC Digital Design and Verification Conventions.

DFG Data-Flow Graph.

DSL Domain Specific Language.

DSP Digital Signal Processing.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

GCC GNU Compiler Collection.

HCL Hardware Construction Language.

HDL Hardware Description Language.

HLL High-Level Language.

HLS High-Level Synthesis.

HW Hardware.

IDE Integrated Development Environment.

IIR Infinite Impulse Response.

IR Intermediate Representation.

LTO Link-Time-Optimization.

RAM Random Access Memory.

RTL Register-Transfer Level.

SoC System-on-Chip.

SW Software.

VCD Value Change Dump.

Chapter 1
Introduction

1.1 Motivation

With the increasing focus on power consumption and small design-size, hardware manufacturer are forced to develop their products with these parameters in mind. Architectural exploration of hardware plays a vital role in the process of creating integrated circuits with the best trade-offs between speed, area, and power consumption for a given specification. The process of architectural exploration is a tedious and time-consuming process, involving many steps. During the exploration, a number of hardware architectures are built and evaluated based on minimum performance requirements and worst-case operational scenarios. By generating a large number of designs with great diversity, a satisfactory result can be achieved. The number of architectures that can be evaluated is limited by available time and resources. High-Level Synthesis (HLS) is a compelling alternative to shorten this process. By reducing the time for creating each design, the number of evaluated designs can be increased, with the potential of generating far more diversity between the architectures than what would ever have been possible by parametrized Register-Transfer Level (RTL).

On the left side of figure 1.1 a typical design process for a Digital Signal Processing (DSP) application is shown. On the right side, the same design process is shown, using a HLS-based framework. It can easily be seen that the effort the designer has to put into the process is reduced with the second alternative.

The thesis will look at the implementation of a framework for architectural exploration of digital hardware, targeted for Application-Specific Integrated Circuit (ASIC) implementation. The ultimate goal is to create a framework that automatically explores a wide variety of architectural variations and presents the best alternatives with regards to a given design goal or constraints.

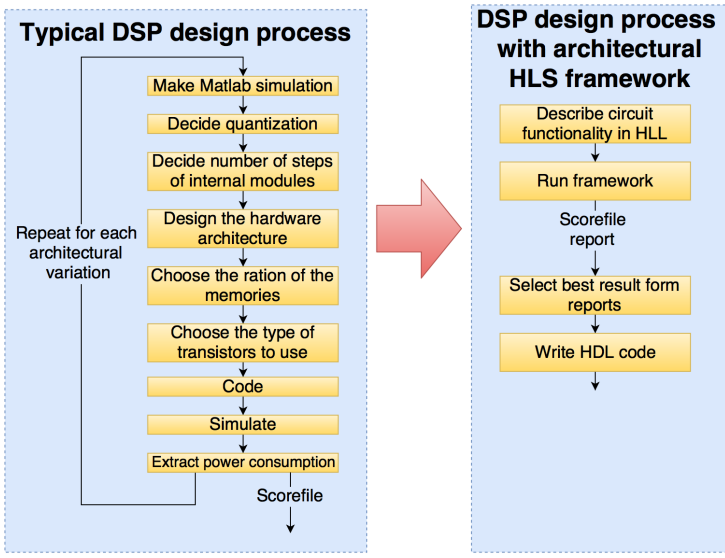


Figure 1.1: Typical DSP design process compared to HLS-framework.

1.2 Previous work

In my specialization project [13], conducted during the autumn of 2015, I explored the academic open source HLS-tool *LegUp*. This tool has a maturity not before seen in an academic HLS-tool, and that it is open-source makes it appealing for the concept of a framework for architectural exploration of hardware. LegUp provides ANSI-C to Verilog high-level synthesis, but their focus is targeted towards implementation on Field-Programmable Gate Array (FPGA) architectures. The official target support of the output is limited to a few boards from the FPGA manufacturer Altera, and beta-support for a single board from Xilinx. This thesis will target ASIC implementations. The findings from [13] was that there are some issues with the original version of LegUp, limiting its usability for the desired framework. The issues are mainly related to input and output of the generated modules, structure of memory management, and size of signals. A framework for architectural exploration of hardware, using HLS, was proposed in [13]. An illustration showing the tool- and information-flow of the framework is shown in figure 1.2.

1.3 Project objectives

The initial goals of the specialization project were found to be a bit exaggerated. For this Master's thesis it was decided to focus on a smaller part of the ultimate goal, to get the necessary basics of the HLS-tool working well, before proceeding with the framework. The main goal of this thesis is therefore to resolve the issues encountered

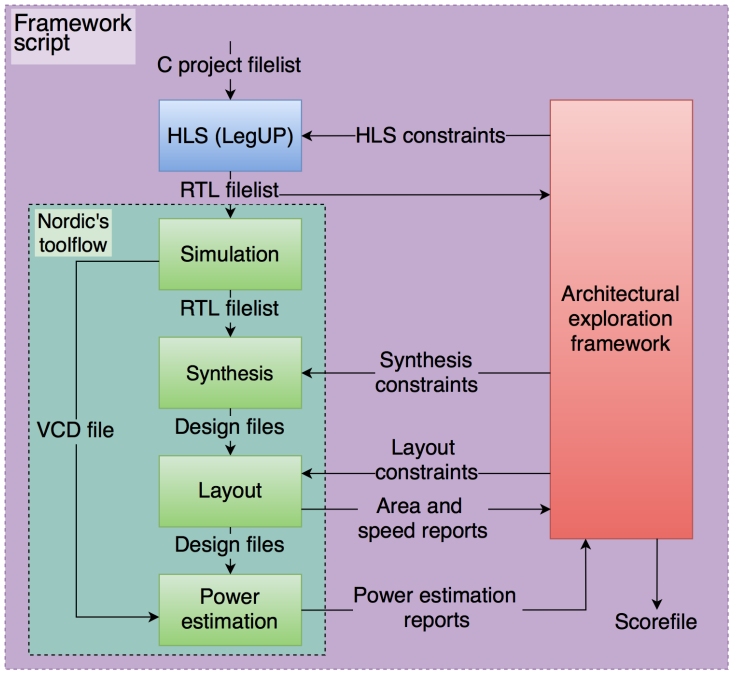


Figure 1.2: Proposed framework-solution [13].

during the specialization project. It is not know if all issues can be resolved, or how time consuming it will be. Other objectives are therefore added in a prioritized order:

1. **Explore approaches**
   Two possible approaches towards resolving the issued, were described in [13]. The first step of this thesis will be to explore both these alternatives and look at positive and negative sides of each method. The outcome of this objective will affect the rest of the work with this thesis, making it an important decision. All aspects of the two approaches must therefore be taken into consideration before making a choice.

2. **Resolve issues**
   For LegUp to be usable in a framework for architectural exploration, it is vital that the tool is adapted to generated Verilog suitable for ASIC implementation. This objective is thought to be the most time-consuming, and its outcome is very uncertain. However, if completed successfully, the use-space of LegUp can be extended to other concepts. LegUp's architecture is, like the input language C, quite memory-bound. Random Access Memory (RAM) modules, memory controllers, and pointers are used for many things where a simple

signal could have given the same result. It should be looked into if this memory-architecture can be changed by de-referencing pointers or turn memory elements into generic signals. A proper way of handling inputs and outputs should also be implemented, to avoid being limited to a certain amount of ports on the generated designs.

3. **Create framework**
   When the issues have been resolved, the work with creating a framework for architectural exploration can be started. The framework will be based on the flow shown in figure 1.2, using various scripts and programs to run the tool-flow, generating constraints, and creating scorefiles. The framework should be easy to use and ideally be able to run without any interactions with the user.

4. **Proof of concept**
   To verify and illustrate the concept in action, a proof of concept will be created. By creating one or more reference designs which will be run through the framework, it is expected to get a wide variety of generated designs with varying results in terms of area, power consumption, and performance. The reference design will also be implemented directly in Verilog Hardware Description Language (HDL), to compare and calculate the overhead of the HLS-generated designs.

5. **Evaluation**
   Based on the results from the conducted proof of concept, a re-evaluate of LegUp's usefulness in a framework for architectural exploration of digital hardware, will be conducted. This evaluation will be based on the deviation of the results among the generated designs, as well as the overhead compared to the design written in Verilog. Other aspects can also be considered, like how well the adaption of LegUp is performed and how well the generated Verilog HDL synthesize for ASIC architectures.

6. **Techniques for reducing overhead**
   The typical overhead of HLS-tools are in the range of 30-40%. One of the initial objectives of this concept included the integration of Nordic Semiconductor's coding style and practices, the Digital Design and Verification Conventions (DDVC) [29], into LegUp's Verilog libraries. This include things like interfaces, parameters, naming conventions, power/clock domains, etc. It is assumed that this can give a large reduction of the overhead generated by the HLS-tool, when integrated into Nordic Semiconductor's existing modules.

1.4 Contributions

The intentions of this work have been to create an adapted version of the open source HLS-tool LegUp, to make it more suited for generating Verilog targeted towards ASIC architectures. It was also time to create a framework for architectural exploration of digital hardware, and to conduct a proof of concept study.

The following list summarize the contributions made through this thesis:

- An adapted version of LegUp has been created. The adapted version support features that is important for implementation towards ASIC architectures. This include the possibility of having multiple inputs and outputs in the generated modules, the inputs and outputs can be streaming, eliminating the need for stopping and starting the module for each run, and an improved method of generating testbenches that include all signals and desired testcases.
- A framework for architectural exploration of digital hardware has been developed. This framework can generate a large number of architectural variations with great diversity. Area, power and performance information will automatically be extracted from each design, allowing the designer to choose the best architecture for further implementation.
- Using a FIR-filter reference design, a proof of concept study has been conducted, showing that the framework can be used for architectural exploration of digital hardware.
- LegUp's usability in a framework for architectural exploration of digital hardware has been evaluated, based on results from the proof of concept study and the performance of the adapted version of LegUp.

1.5 Method

The work performed in this thesis is based on multiple research methods. Before the problem could be solved, a study of the architecture and structure of LegUp had to be conducted, to understand the connections and information-flow in the tool. This study was primarily carried out during the previous project [13], but also continued into the work with this thesis. A plan for how to resolve each of the issues at hand was devised and discussed before being carried out, to ensure a good solution. The problems at hand requires in-depth knowledge of the libraries in LegUp, but when the source of the issue had been located, fixing the issue was based on trial and error. By replacing a piece of code with some other solution, a new output can be generated and evaluated. This process is repeated until the issue is resolved. The creation of the framework is based on the idea proposed by Isael Diaz. A study of architectural

exploration and HLS-concepts had to be conducted before building the framework, to make sure the output would have the desired diversity. An experimental study of the usefulness of the created framework was conducted as a proof of concept, to check if the initial hypothesis holds. By running a reference design through the framework, a large amount of data was reported. The data was analyzed to draw the conclusion about the hypothesis.

1.6 Overview of the thesis

In general, this thesis is divided into 8 chapters, each presenting one or more of the project objectives described above, in addition to appendix. In chapter 2, the background and theory required to understand the rest of the thesis is described. Point one and two from the list above is described in chapter 3. Chapter 4 uses a design example to present a thorough description of the information-flow in LegUp and the other tools used in the framework. In chapter 5 the third objective, the process of creating a framework, is described. The fourth objective, to create a proof of concept, is presented in chapter 6. The evaluation of the proof of concept results, corresponding to the fifth objective, as well as a discussion of LegUp in general, with focus on its usefulness in the created framework, has been presented in chapter 7. Finally the work is summarized and concluded in chapter 8. Chapter 8 also include a section of future works, describing aspects that will be interesting to look into more detail at in an eventual continuation of this project. Appendix include code-listings of designs and implementations, that are described and discussed in the main chapters.

Chapter 2
Theory and background

Some theory and background is needed to get a thorough understanding of the material in the following chapters. Some parts of this background chapter were written as part of the specialization project [13], but it is included here to allow the report to be a freestanding document. Some sections have been extended to add a deeper level of understanding to some of the described concepts, compared to what was presented in the previous report. Some information from section 3.3 of the *Methodology*-chapter has also been included in section 2.6 of this report, as it describes part of the same tool-flow used here.

In the early days of digital hardware design, gate design and layout were performed by hand. With the rapid growth in the numbers of transistors per digital chip-design, this method quickly became too time-consuming and the need for new and more automated design methods rose. RTL-design using HDL has long been the standard in digital hardware design. With the increasing demand for low power and small area in large System-on-Chip (SoC) designs with multiple billion transistors, this methodology is no longer sufficient if hardware manufacturers want to hit the window of opportunity with their state-of-the-art product.

2.1 High-Level Synthesis

HLS is not a new concept as it was introduced in research papers in the late 1970s and further researched and developed in the 1980s and early 1990s [23]. The available commercial HLS tools have not been providing the necessary performance and benefits over HDL development for major hardware development companies to adapt this methodology until recently. The concept of HLS starts with a functional specification of the circuit described using a higher abstraction level, often a High-Level Language (HLL). A tool uses target architectural model libraries and design constraints to transform this specification into hardware, represented as a RTL or HDL-model. The typical HLS-flow is shown in figure 2.1 and each of the transition-steps is described in

the below subsections. The input libraries contain information on available hardware resources with power, area, and delay models for the target architecture.

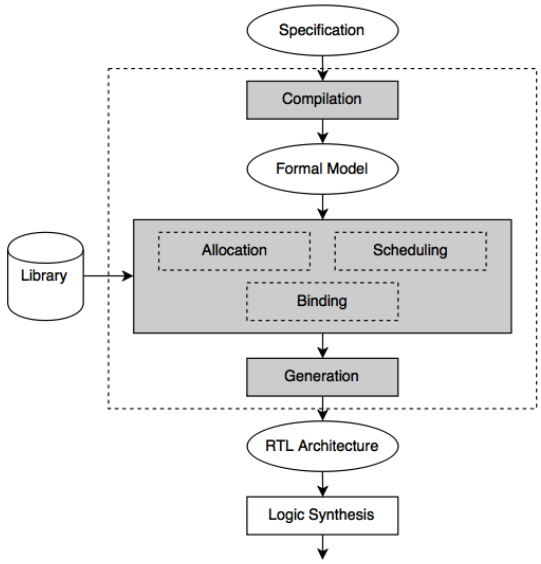


Figure 2.1: Information flow in a typical HLS-tool [8].

Compilation

The first step of HLS is to compile the functional specification into a formal model. This model can vary between different tools, and can be either a specific representation language or a graphic representation of the flow. The formal model is decided by the developers of the HLS tool.

Allocation

Necessary hardware resources, such as functional units, storage-, and connectivity-components needs to be selected from a given RTL component library in order to satisfy the specification and design constraints. Some HLS tools can also add more resources in the scheduling and binding tasks, if this is needed to meet given constraints.

Scheduling

Scheduling arranges all operations in an optimized sequence so that variables are read from sources and brought to the input of the correct functional unit for execution and to the destination afterwards. The scheduler takes all dependencies into account when scheduling the operations, in order to get the most efficient result, as some

operations can be executed in parallel if no dependencies exist and there is available resources. Operations can be scheduled to finish in one, or take multiple clock-cycles, and operations can also be chained to eliminate the need for storing the result between operations, and to reduce the total number of cycles needed.

Binding

In the binding task, all clock-cycle-crossing variables, operations, and transfers are bound to a free resource, in the time-frame when it is scheduled. Non-overlapping or mutually exclusive variables can be bound to the the same storage unit, and operations can be bound to the best optimized functional unit if multiple alternatives are available. Each transfer from component to component, either storage or functional unit, needs to be bound to a connection unit, such as a bus or a multiplexer.

RTL Generation

The generated RTL usually consists of two parts, a control-unit and a data-path-unit. The control-unit is often implemented as a Finite State Machine (FSM), which set control-signals to the data-path, and controls the current and next-state of the system. The data-path contains storage-, functional-, and connection-units. An example of this division is shown in figure 2.2. Depending on the intensiveness of the binding

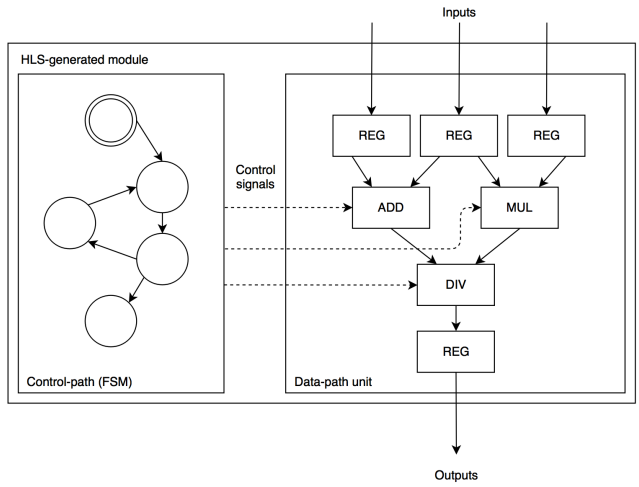


Figure 2.2: Typical division of control and data-path in the generated RTL from HLS.

step, the output RTL can be tightly or loosely bound to the available resources. If an operation is not bound to a specific unit, it is up to the following logic synthesis of the RTL to bind the operations to available resources. The different types of RTL output are illustrated by the following example. $a = b * c$ executing in state n:

Without any binding:

```
state (n): a = b * c;
go to state (n + 1);
```

With storage binding:

```
state (n): S(1) = S(2) * S(3);
go to state (n + 1);
```

With functional-unit binding:

```
state (n): a = MUL1 (b, c);
go to state (n + 1);
```

With storage and functional-unit binding:

```
state (n): S(1)=MUL1 (S(2), S(3));
go to state (n + 1);
```

With storage, functional-unit, and connectivity binding:

```
state (n): BUS1 = S(2); BUS2 = S(3);
BUS3 = MUL1 (BUS1, BUS2);
S(1) = BUS3;
go to state (n + 1);
```

A loosely bound RTL gives the synthesis-tool the flexibility to optimize the unit binding to updated timing estimates, delays, and loads given by the layout and floor-planning tools.

2.2 LegUp

The HLS tool used in this project is called LegUp [6]. LegUp is an open-source academic tool developed at the University of Toronto, Canada. LegUp's goal is to *"allow researchers to experiment with new HLS algorithms without building a new infrastructure from scratch"* and their long-term vision is to *"make FPGA programming easier for software developers"* [4]. LegUp takes American National Standards Institute (ANSI)-C as input and generates synthesizable Verilog HDL as output. The developers of LegUp have primarily focused on support for a variety of FPGA boards from manufacturer Altera, but in the latest version (4.0), beta support for Xilinx devices [20] and possibility to configure the tool to generate generic Verilog to target other FPGA vendors or even ASIC through use of generic dividers [18], has been introduced. The big advantage of LegUp compared to similar, commercial tools, is that it is open-source and therefore can be configured to target different

architectures. The RTL and HDL generating part of the tool can be modified or replaced to fit the programmers needs. Since LegUp, in its unmodified form, target FPGA devices, it supports three different synthesis flows; pure-Software (SW), hybrid, and pure-Hardware (HW). The two first synthesis flows will implement a TigerMIPS [24] soft processor, which will run part of the C code. The partitioning of SW and HW in the individual modules are described in table 2.1. It is the pure-HW flow that will be the focus of this project.

Table 2.1: HLS-flows supported by LegUp and partitioning between SW and HW

Flow	Functions run in hardware	Functions run in software
Pure-SW	None	All
Hybrid	Specified hardware-accelerated functions	All other functions
Pure-HW	All	None

2.2.1 Producing Verilog Output

LegUp is made up of two components; a frontend pass and a target backend pass to the LLVM compiler infrastructure. The information flow in LegUp, shown in figure 2.3, follows the same principle as the information flow described in section 2.1. The LegUp LLVM frontend takes LLVM Intermediate Representation (IR) compiled

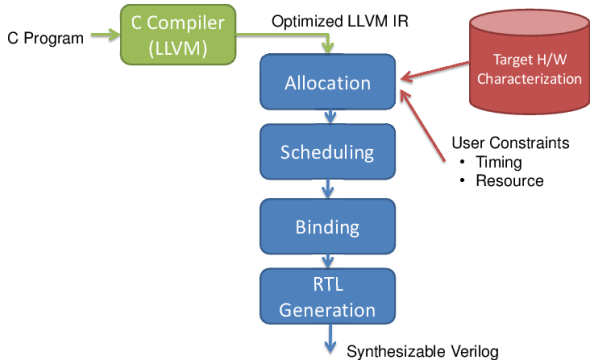


Figure 2.3: Information flow in LegUp [19].

by clang, a C frontend for LLVM, as input and links in custom written functions like memcpy, memset and memmove, which do not exist in hardware, but that LLVM assumes exist in the C library. The LegUp backend pass performs allocation, scheduling and binding as described in section 2.1. In the next step, RTL-module objects that represents the final hardware circuit are generated from each LLVM

instruction. Ultimately, Verilog code corresponding to each of the RTL-modules is output to a file.

The allocation, scheduling, and binding in LegUp is performed based on information about available resources and timing information about the specified target FPGA-board, in addition to user-defined constraints and setting. The available information about the FPGA-boards allows for precise scheduling and binding to the available resources. Since the implementation of ASIC designs are quite different from the architecture and implementation of designs on FPGAs, the resource and timing information will not be as easily obtained for the target architecture.

2.2.2 Classes

In LegUp there are some predefined classes that is important for the understanding of the description of adapting LegUp, presented in chapter 3. The following subsections will describe some important information about these classes in more detail. The full class descriptions can be found in the *LegUp Namespace Reference* [9].

RTLModule

The RTLModule class models a hardware RTL module. The class stores information about all ports (inputs and outputs), signals, parameters and sub-modules. Each function declared in the C-code transforms into a RTLModule object. Each function that is called from the function will be added as a sub-module to the RTLModule object, meaning a module instantiation will be added to the module. Important member-functions of the RTLModule class are:

getName()
: Returns a string containing the name of the RTLModule, i.e. "main" for the module generated by the *main*-function in the C-program.

find(std::string signal)
: Takes a string containing a signal name as parameter and returns a pointer to the RTLSignal in the RTLModule with that name.

addParam(std::string name, std::string value)
: Adds a parameter to the module. The function returns a pointer to the generated RTLSignal object.

addIn(std::string name, RTLWidth width)
: Adds an input-port to the module. The function returns a pointer to the generated RTLSignal object.

addOut(std::string name, RTLWidth width)
: Adds an output-port to the module. The function returns a pointer to the generated RTLSignal object.

addRegOut(std::string name, RTLWidth width)
Adds a registered output-port to the module. The function returns a pointer to the generated RTLSignal object.

addReg(std::string name, RTLWidth width)
Adds a register signal to the module. The function returns a pointer to the generated RTLSignal object.

addWire(std::string name, RTLWidth width)
Adds a wire signal to the module. The function returns a pointer to the generated RTLSignal object.

addModule(std::string name, std::string instName)
Adds an instantiation of another module to the module. The function returns a pointer to the generated RTLModule object.

RTLSignal

The RTLSignal class represents the signals within an RTLModule. Both internal signals, port signals and condition signals are all modelled using the RTLSignal class. Important member-functions of the class are:

getName()
Returns a string containing the name of the RTLSignal, i.e. "clk" for the clock signal.

getType()
Returns a string describing the signal type. The type can be *reg*, *wire*, *input*, *output*, or *output reg*.

getNumDrivers()
Return the number of driving RTLSignals.

getDriver(unsigned i)
Returns a pointer to the i-th driving RTLSignal.

getCondition(unsigned i)
Returns a pointer to the condition signal of the i-th driving RTLSignal.

addCondition(RTLSignal *cond, RTLSignal *driver)
Adds a conditional driver. If the RTLSignal *cond* is true, the RTLSignal *driver* drives the signal.

connect(RTLSignal *s)
Connect this signal unconditionally to another RTLSignal.

getWidth()
Returns a pointer to a RTLWidth object, describing the width of the RTLSignal.

isOp()
Returns true if the RTLSignal is an RTLOp object.

RTLOp

The RTLOp class is a subclass of the RTLSignal class, representing an operation with one, two or three operands. Each operand is a RTLSignal. The operation can be an arithmetic operation like addition, subtraction, multiplication, or division, and it can also be logical operations like AND, OR, and XOR, or even comparison operations like equal, not equal, less than, less than or equal, greater than, and greater than or equal. The whole list can be seen in the class reference [10]. A RTLOp object modelling an AND operation of two operands, operand1 and operand2, will in Verilog correspond to the operation "`operand1 & operand2`". Some important member-functions are:

getOperand(int i)
: Returns a pointer to i-th operand of the RTLOp object.

getNumOperands()
: Returns the number of operands of the RTLOp object.

setOperand(int i, RTLSignal *s)
: Sets the i-th operand to the RTLSignal *s*.

RTLWidth

The RTLWidth class represents the bitwidth of a RTLSignal. An RTLWidth is defined by high and low bits, for instance 31,0 for a 32 bit signal. This will transform into "`[31:0]`" in Verilog.

RAM

The RAM class models RAM modules in LegUp. Whenever a variable is loaded or stored, a RAM module is generated to handle the loads and stores. The RAM objects can be divided into two scopes; LOCAL and GLOBAL. A local RAM object is local to a given function and cannot be accessed by other functions. A global RAM object will be implemented in a global memory controller. All modules that use the variable can connect to the RAM via the memory controller. Some important member-functions are:

getName()
: Returns a string containing the name of the RAM module, i.e. "main_0_1" for the RAM module generated for the first parameter to the main function declared as volatile (output parameters) in the C-code.

isROM()
: Returns true if the RAM is read-only.

getScope()
: Returns if the RAM is in the local or global scope.

2.2.3 Constraints

Constraints is an important part of LegUp, and it is also used extensively in this project. The constraints are used for setting design goals and limitations on design, and to specify how the HLS-flow will be executed. Constraints play an important role in this concept, as the idea is to generate multiple designs that can be compared in terms of area, performance, and power consumption. For the designs to be different, varying constraints are used for generating the designs. All available constraints are described in the constraint manual [17], but the ones used in this project are described in table 2.2. Some constraints are considered *required*. These constraints must be set for the generated design to be compatible with the tool-flow. *HLS constraints* are used for getting different Verilog-outputs from LegUp. Other constraints from the constraint manual can also be used, but these were selected for this project as their description indicate that they can affect the architecture of the output.

2.3 LLVM

LLVM [16], formerly Low-Level Virtual Machine, is a compiler framework that was originally developed as a research infrastructure to investigate dynamic compilation techniques for static and dynamic programming languages, at the University of Illinois in 2000. It is now a open-source project with many contributors from both industry, research groups and individuals, and it is used by companies like Apple in their Xcode Integrated Development Environment (IDE) [21] and Sony for their PS4 developer toolchain [28]. LLVM support a large number of frontends for programming languages, including Clang [7] which support C, C++, Objective-C, and Objective-C++, and is compatible with GNU Compiler Collection (GCC). It also supports a large number of backend target architectures. Figure 2.4 shows how different source languages can be input to the frontend compilers of LLVM, which translate the source into an IR. The IR is then optimized using LLVM's optimizer. At this stage, different source languages can be linked together, and even object files compiled using standard GCC can be linked at this stage. The optimized IR is then translated into the target architecture by the backend.

2.3.1 Intermediate Representation

LLVM use a human readable, assembly-like, strongly typed RISC instruction set as the IR, with support for an infinite number of temporary registers of the form %0, %1, etc. LLVM can also output a dense bitcode format of the IR for serialization. Conversion between the bitcode-format and the human-readable format, and vice versa, can be done with the commands "`llvm-dis`" and "`llvm-as`", for dis-assembly and assembly.

Required constraints

Parameter name	Description	Required value
DIVIDER_MODULE	Use generic divider module rather than Altera primitive	*generic*
EXPLICIT_ LPM_MULTS	Use Altera primitive multiplier rather than Verilog multiply operator (*)	*0*
INFERRED_RAMS	Use Verilog inferred RAMs rather than Altera altsyncram modules	*1*
INFERRED_ RAM_FORMAT	Select format of inferred RAMs. Altera: multiple always blocks, Xilinx: same always block	*xilinx*
LOCAL_RAMS	Infer all RAMs as local RAMs rather than global RAMs. RAMs being accessed by multiple functions will override this setting	*1*
VSIM_NO_ASSERT	Disable triple-equality assertions. This causes simulation to fail	*1*

HLS constraints

Parameter name	Description
SDC_NO_CHAINING	Schedule each operations into a separate clock cycle
MB_MINIMIZE_HW	Run LegUp-pass that tries to minimize signal sizes
CASE_FSM	Use case-statements in FSM rather than If-Else
PIPELINE_ALL	Enable pipelining for all loops, regardless of loop-label
ENABLE_ PATTERN_SHARING	Turn on resource sharing for patterns in Data-Flow Graph (DFG)
DUAL_PORT_ BINDING	Use dual-ported on-chip memories

Table 2.2: Description of constraints used in this project

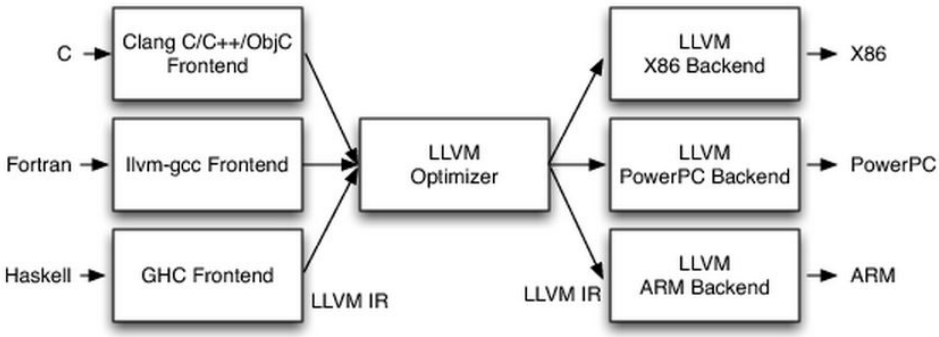


Figure 2.4: LLVM's three-phase compiler structure [15].

Some parts of the LLVM IR will be described in chapter 4. The whole language is too large to be fully explained here, but interested readers can read more about the syntax in the *LLVM Language Reference Manual* [14].

2.4 Alternative hardware design methods

HLS is not the only alternative to HDL-languages, if you want to design digital hardware at a higher level of abstraction. The following subsections will shortly describe two alternative approaches to digital hardware design.

2.4.1 Chisel

One interesting approach to designing hardware with a higher level of abstraction, is the Chisel Hardware Construction Language (HCL) [2], developed at UC Berkeley. HDL languages like VHDL and Verilog, were originally designed as simulation languages and later adopted as a basic for synthesis. Chisel, on the other hand, was created as a HCL and is thus *synthesizable by construction*. This entails that no conversion from C, or other HLL, into gates is performed, only generation of generic low-level Verilog with no overhead. Chisel is a Domain Specific Language (DSL) built on Scala [25] with its own syntax, but Scala syntax can also be used to get even greater abstraction in your design. A big advantage using Chisel is its high simulation speed, using C++-based cycle-accurate software simulators.

2.4.2 Functional programming

Functional programming is a relatively different method of hardware design, as it consists only of mathematical functions and immutable data. Two examples of hardware design using functional programming is CλaSH [1] and Lava [3]. Both Lava and Cλash are compilers for the functional programming language Haskell [12], but

while Lava is an embedded DSL like Chisel, with its own syntax, Cλash use Haskell syntax and semantics, and use a static analysis approach towards synthesis.

2.5 Power dissipation in CMOS circuits

The power dissipation in CMOS circuits can be divided into three categories [26], *dynamic power*, *short-circuit power* and *leakage power*. This gives a total power dissipation of:

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage} \tag{2.1}$$

Figure figure 2.5 shows the distribution of the power components of the CMOS circuit. Each component is described in more detail in the following subsections, where *switching power* corresponds to $P_{dynamic}$, *internal power* corresponds to $P_{short-curcuit}$, and *leakage power* to $P_{leakage}$.

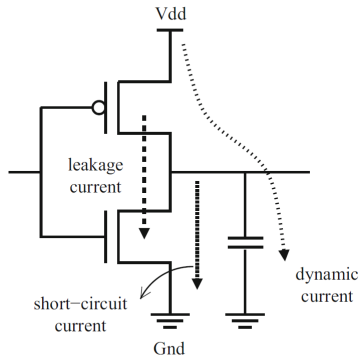


Figure 2.5: Power dissipation components distribution [26].

2.5.1 Switching power

Whenever a signal changes the logic state from 0 to 1, the load capacitance is charged by the power supply. The power dissipated during this process is called *switching power*. Half the energy drawn from the power supply needed to charge the capacitance, is dissipated as heat in the process. The *switching power* depends on the frequency of the switching, the switching factor of gates, and the load capacitance, in addition to the supply voltage.

2.5.2 Internal power

The *internal power* is the power used to charge and discharge the internal capacitance of the circuits, whenever a pin changes its logic state. A large part of the *internal power* is the short-circuit power. In the short time when both the pMOS and nMOS transistor of the CMOS circuit is *on*, a current will be drawn from the source V_{dd} to *Gnd*, through the short-circuit that will occur.

2.5.3 Leakage power

Whenever the circuits are turned *on*, a small leakage current will be drawn from the gates. The leakage power is mostly caused by sub-threshold currents and reverse biased diodes in the circuits. The leakage current increase when the technology shrinks, making leakage a bigger problem today than before.

2.6 Tool-flow

This section will describe all the tools that are used throughout this thesis, as well as the connection and data-flow between the different tools. This flow is based on the standard tool-flow used at Nordic Semiconductor, and it include some parts adapted from the "*automated area and power estimation tool-flow*" created by Joar Talstad for his Master thesis [35]. Most of the tool-flow is based on scripts and Makefiles that can be run from a Linux shell, but there are also some GUI-tools available that will be mentioned briefly in chapter 4. The following subsections will describe the different sections of the tool-flow in detail. The flow in LegUp will not be described here, as this is covered above and will be presented in more detail in chapter 4.

2.6.1 Simulation

Simulation is run to verify the correctness of a design and to help detect and eliminate potential bugs. In this project, the simulation tool also generates a Value Change Dump (VCD)-file, *designname.vcd*, showing switching activity during simulation. This file is used in the power analysis tool later in the flow, to get a realistic input of the amount of switching in the design. Simulation is performed using the tool ModelSim for Questa-64, version 10.2b 2013.05 [11]. Simulation is executed by calling the script *RUN_ALL*. The RTL-design filelist and a file containing a testbench module must be specified in the filelist found in the *sim/tb/*-directory. This is used as input to the simulation tool.

2.6.2 Synthesis

Synthesis translates a RTL-design written in a HDL-language, like Verilog or VHDL, into a netlist for a specified target library. The tool used for synthesis in this thesis is

Synopsys Design Compiler, version I-2013.12-SP2 [32]. A cell library describing 180nm technology is used as the target architecture. A Makefile is used to start synthesis, and the command `make compile` runs the full synthesis. The netlist generated by synthesis is found in the file *designName.mapped.v* in the *result*-directory. This netlist is used as input for the layout-tool. Synthesis generate reports showing area-estimates, register count, critical path and static power estimates for the design. As the design will be processed further through the tool-flow, these reports are not that accurate and hence not that useful.

2.6.3 Layout

Layout translates the netlist generated during synthesis into a chip layout. The tool used for layout in this project is Synopsys IC Compiler, version L-2016.03-SP1 [33]. A Makefile is used to start layout, and the command `make outputs_cts` runs the correct layout-script. Layout produces a new netlist-file, stored in the file *designName.output.v* in the *result*-directory. This netlist is used in the power analysis tool for estimating power consumption. Layout generate reports about area and critical paths, stored in the *reports*-directory. These results are more accurate, as they were gathered from the actual chip layout.

2.6.4 Power analysis

Power analysis is performed to get an early indication on how much power the final chip will be consuming. The tool used for power analysis in this thesis is Synopsys Primetime, version K-2015.12-SP3. To get accurate power estimates, the switching activity file generated during simulating is used together with the netlist output from layout. The conclusion from [35] was that this method provides accurate results and is well suited for making RTL-design trade-offs based on power consumption in multi-voltage designs. Power analysis is run on five different power scenarios, each giving a separate result for each of the three power dissipation categories described in section 2.5. The reports are stored in the *reports*-directory.

2.7 Reference design

This thesis will look into whether or not LegUp can be used as the HLS-tool in a framework for architectural exploration of hardware. In order to get some output from LegUp that can be compared towards each other, a reference design must be created. The design will be used in the proof of concept, described in chapter 6, and should be something that can be implemented both in C and Verilog. The design should also be simple to implement and verify. In [13], two reference designs were implemented; a FIR-filter and a SAP-1 architecture. The FIR-filter will be used as the reference design in this project, as this is a regular structure that easily can be

implemented and verified. The SAP-1 architecture would have been a interesting second reference design, as it consists of a FSM, just like the output from LegUp. Unfortunately, this architecture has too many design-parts that will be incompatible with the framework. It has therefore been decided to leave this design out of this thesis.

2.7.1 FIR-filter

Finite Impulse Response (FIR)-filters are together with Infinite Impulse Response (IIR)-filters, the two categories of linear time-invariant systems, used in digital signal processing application. The impulse response of a FIR-filter is zero outside some finite time interval. A general FIR-filter can be described by the differential equation [27]:

$$y(n) = \sum_{k=0}^{M-1} b_k x(n-k) \tag{2.2}$$

or by the system function:

$$H(z) = \sum_{n=0}^{M-1} b_n z^{-n} \tag{2.3}$$

The impulse response for a FIR-filter is given by:

$$h(n) \triangleq \begin{cases} 0, & n < 0 \\ b_n, & 0 \leq n \leq M-1 \\ 0, & n > M \end{cases} \tag{2.4}$$

From eq. (2.2) and eq. (2.4) we get the discrete convolution equation:

$$y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k) \triangleq h(n) * x(n) \tag{2.5}$$

Figure 2.6 shows the direct form representation of a N-order FIR-filter with $N+1$ taps. The figure shows that a FIR-filter requires N memory elements, N adders and $N+1$ multipliers.

Even though the process of designing a FIR-filter might not be a trivial task, the implementation of an already designed filter is simple. As seen from eq. (2.5), the filter can be described by the convolution formula, which implies that the filter can be implemented as convolution of the input function *x(n)* with the impulse response function *h(n)*.

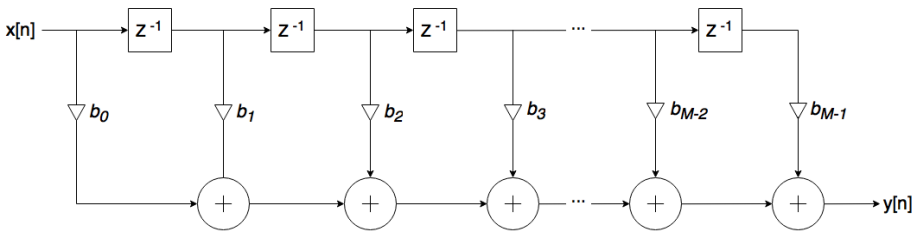


Figure 2.6: Direct form representation of a N-order FIR-filter.

Chapter 3
Adapting LegUp

The main focuses of this thesis has been to resolve the issues encountered in [13], to make LegUp able to generate Verilog more suited for ASIC implementation and synthesis. This chapter will describe the process of resolving these issues and other alterations that have been added to simplify the creation of a framework for architectural exploration of hardware.

3.1 Approach

In the future works section of [13], two different approaches to resolving the issues were proposed; post-processing and pre-processing. Both approaches have been explored, but the majority of solutions are based on the pre-processing alternative. The two following subsection will present the two approaches and give some reasoning to why one is preferred over the other.

3.1.1 Post-processing

With the post-processing approach, the idea is to alter the Verilog-code after it is generated, to make it more suitable for ASIC implementations. This approach is easy to work with, as we can concentrate on a single file, the output Verilog file. The drawback of this approach is that you only have the information available in the Verilog file at hand, making it hard to add functionality to the tool.

There exist multiple parser tools for Verilog, for instance Verilog-Perl from VeriPool, a Verilog parser library for Perl [30], and pyverilog, a Hardware Design Processing Toolkit for Python [34]. These tools can be used to parse the Verilog file, to build module, signal, and port hierarchy, and easily add, alter, or remove objects.

3.1.2 Pre-processing

The pre-processing approach involves changing the libraries in LegUp that perform HLS operations like allocation, scheduling, RTL-generation and Verilog printing. This requires deep knowledge of the libraries and its connections, to find a good way to change the output. The large libraries is the main drawback of this approach. As LegUp is open-source, the possibilities of this approach are endless, but getting the necessary knowledge of the libraries takes time.

3.1.3 The used approach

As it looked like the easiest solution, the post-processing alternative was explored first. However, it was soon realised that the things that could be done easily with this approach, also could be done quite easily with the pre-processing approach. Some larger issues, for instance assigning values to outputs, were not easily solvable using the post-processing method. The focus was therefore directed towards the pre-processing alternative. One advantage of this approach is that the original functionality of LegUp can be kept, while adding new functionality. The switching between original and altered versions are done using TCL-parameters. The post-processing method was used at a later stage, but then on the LLVM IR-code rather than the generated Verilog.

3.2 TCL commands

LegUp uses TCL commands for setting constraints and configuring the HLS-flow. In order to keep the original implementation of LegUp, and to provide additional functionality, some new commands were added. New TCL-parameters can easily be added to LegUp by adding the parameter name to the array *validParameters* and increasing the parameter *NUM_PARAMETERS* in the file *LegupConfig.cpp*. The value of the parameter can then be read using the function call:

```
LEGUP_CONFIG->getParameter("parameterName")
```

to get a string, or

```
LEGUP_CONFIG->getParameterInt("parameterName")
```

to get an integer. *LegupConfig.h* must be included to get access to LEGUP_CONFIG. The most common use of TCL-parameters is to check whether a parameter is set, and perform some action based on this. Parameters can also be used to set values of variables. An example could be a parameter that decides if a designated top-module will be generated or not.

The parameter is defined by adding the following code to the constraint file:

```
set_parameter PRINT_TOP_MODULE 1
```

The parameter can then be used to decide if the top-module should be printed:

```
if(LEGUP_CONFIG->getParameterInt("PRINT_TOP_MODULE") {
  printTop();
} else {
  printVerilogWithoutTop();
}
```

Another example is to use a parameter to set the name of the top-module. This can be used for naming the top-module, or to select top-module in the simulation-settings.

```
set_parameter TOP_MODULE_NAME "moduleName"
```

```
std::string topModuleName = "top"; // Default name
if(LEGUP_CONFIG->getParameter("TOP_MODULE_NAME") {
  topModuleName = LEGUP_CONFIG->getParameter("TOP_MODULE_NAME");
}
```

In the second example, the *getParameter()* function will return false if the parameter is not set.

Other TCL-commands can also be defined by adding the following line to the file *LegupTcl.cpp*:

```
Tcl_CreateCommand(interp,
                  "set_custom_main_function",
                  set_custom_main_function,
                  legupConfig,
                  0);
```

Here the second parameter is the TCL-command and the third parameter is the handler function that will be called when the TCL-command is encountered. In the handler function, arguments from the constraint file can be used to configure LegUp. As multiple arguments are supported, more advanced configurations can be performed with this alternative. The parameters that has been added to LegUp is described below.

ASIC_IMPLEMENTATION

This parameter is used to distinguish between the original version of LegUp and the altered version developed in this thesis. If this parameter is set, all

extra features described in the following subsections will be applied to the generated design. If the parameter is not set, the unaltered edition of LegUp will be used to generate the output.

set_custom_main_function

This parameter can be used to define inputs and outputs in the *main*-module, as described in section 3.5.2. As this is not a simple TCL-parameter, it takes multiple arguments. The format of the input should be:

```
portDirection portSize portName
```

An example of declaring two inputs and two outputs in the *main*-module could be:

```
set_custom_main_function
    input   7:0 inSignalA \
    input  31:0 inSignalB \
    output 31:0 outSignal \
    output  1:0 outSignalValid
```

ENCLOSING_WHILE_LOOP

Indicating that the *main*-function has enclosing while loop (for streaming inputs/outputs). Will generate *iterationFinish*-signal each time an iteration of outer while loop is finished.

SEPARATE_TB_FILE

Parameter decides if testbench is printed in same file as design or in a separate file. The filename of the separate testbench-file will be *test_main.v*, according to Nordic Semiconductor's naming-convention, but this can easily be changed or made dynamic by setting the parameter *SEPARATE_TB_FILENAME*.

SEPARATE_TB_FILENAME

Take testbench filename as parameter and changes the default filename of the testbench output-file to this name. Will not have any effect if *SEPARATE_TB_FILE* is not set.

TB_TESTCASE_FILE

This parameter provides the filename of a file containing testcases for the testbench. The testcases will be automatically included into the testbench, as described in section 3.9. If the parameter is not set, no testcases will be added to the testbench.

REMOVE_UNUSED_LOCAL_RAMS

By declaring input parameters as *volatile*, a local RAM will be generated in the *main*-module for each output signal we create. These RAMs are not used for

anything useful and can therefore be removed to save area. If set, local RAMs in *main* are removed **only if** the value stored to the RAM is assigned to an output instead.

3.3 Removing top-level and FPGA-specific modules

As described in [13], the output Verilog contains many module declarations not required or wanted in an ASIC implementation. This include the modules *top*, *memory_controller*, *circuit_start_control*, *hex_digits*, *%board%* and *main_tb*. The modules *memory_controller* and *main_tb* are discussed in sections below, but it is also desirable to remove the other modules. Excess modules could easily be removed by parsing the generated Verilog-file, but the output can be easily controlled with the use of TCL-parameters in the VerilogWriter-library of LegUp. When the parameter *ASIC_IMPLEMENTATION* is set, none of these modules are printed to the generated Verilog file.

3.4 Removing memory controller

One of the main issues with using LegUp for ASIC implementations, is that a global memory-controller for passing data between modules, are added to the design. With this architecture, values have to be added to the memory prior to the run, or continuously during the run. This generates additional timing requirements and adds extra logic for handling these operations. Both to decrease the overhead, and to simplify the generated design, it is desirable to avoid this memory controller. A simple solution to this, is to set the parameter *LOCAL_RAMS* to 1. This parameter is already present in LegUp. Setting this parameter will prevent the global memory controller to be generated, as long as there are no variables used by multiple functions (global variables), or pointers that cannot be connected to a single function after points-to analysis. Typically the memory controller will be instantiated in the *top*-module, but as described in section 3.3 this module is removed when the parameter *ASIC_IMPLEMENTATION* is set. This leads to no connections between the *main*-module and the RAM-modules in the global memory controller, resulting in a failing circuit. It is therefore important to check that the global memory controller is not added to the design. This check has been implemented in the framework-script, described in section 5.2. By using the tool *grep* to search for the line "`module memory_controller`" in the generated Verilog-file, the user will be notified if the memory controller is found in the design.

3.5 Declaring inputs and outputs

Each function declared in the input C-code will be translated into a Verilog-module by LegUp. Since LegUp primarily is designed for implementing hardware accelerators for FPGAs, it does not handle inputs and outputs well to and from the *top*-module. In an ASIC implementation, inputs and outputs are essential to most module design and must therefore be easy to implement. In a C-code written for execution on a computer, the input parameters to the *main*-function is defined to be on the form "`int main(int argc, char *argv[])`". This limits the possibility to declare inputs to the module with any data-type. To solve this problem, the flag *-ffreestanding* has to be passed to the clang compiler frontend of LLVM. The compiler will then consider the C-code to contain a freestanding - not a hosted - environment. The types of inputs and return-values defined in the *main*-function will then be of no concern to the compiler. In LegUp, the flag can be passed to the compiler by adding it to the variable *CLANG_FLAG* in the file *Makefile.config*. The solution that would have been used in a hosted environment is to use pointers for input and output parameters, but this would reintroduce the undesired memory controller in the design.

Two different solutions for declaring inputs and outputs are considered and implemented. Both solutions are based on declaring both inputs and outputs as parameters to the *main*-function.

3.5.1 Name prefix

The first solution is to use a prefix to distinguish between input- and output-parameters. The prefix is set to *___out_*, as it is sensible to use a prefix that is seldom used in a variable name. Previously, LegUp assumed all function parameters were inputs, and added the signals to the RTLModule. This has been altered to check the name of the parameter and add it as an output reg if the name starts with *___out_*, otherwise add it as an input. The pseudo-code of how inputs and outputs are handled are shown in algorithm 3.1. Here we assume that i is a function parameter and *rtl* is the RTLModule generated by the *main*-function.

Algorithm 3.1 Adding parameters to a module

```
1: sigName ← i.getName()
2: if sigName.startWith() = "___out_" then
3:     sigName ← sigName.strip(___out_)
4:     i.setName(sigName)
5:     rtl.addOutReg(i)
6: else
7:     rtl.addIn(i)
8: end if
```

The advantage of this method is that it is simple to implement and easy to use, as the user only has to remember the name prefix when writing the functional specification. The name-prefix can also be useful in other sections of the program, as we will see later in section 3.6. The disadvantage is that the name prefix needs to be used throughout the program. It would however be preferable to use a temporary variable in the program until the final value is calculated and ready to be assigned to the output. This will reduce the amount of times the name-prefix must be used. The name prefix will be stripped by LegUp, providing clean signal-names in the final Verilog-module.

3.5.2 TCL-command

The other alternative is to use a TCL-command to define the parameters as input or output. This enables the possibility to also define the size of the signal, but LegUp does not allow setting the size of a signal to a number of bits lower than the size of the defined type. This means that if a parameter is declared as an int in the C-program, LegUp does not allow for setting the RTLWidth of the signal to anything below 32 bit.

Inspiration for this method comes from the parameter *set_custom_verilog_function* already present in LegUp. This is used to add custom Verilog functions to the design. The TCL-command *set_custom_main_function* was added, which generates a vector with objects of the class CustomVerilogIO, each describing one input- or output-signal to the *main*-module. By looping over the vector, each parameter can be added to the RTLModule based on this information. This part is quite similar to the above described name-prefix method. As this method does not provide any additional functionality, it is recommended to use the name-prefix method. The name-prefix method is also required together with another alteration described in section 3.6.1.

3.6 Assigning values to outputs

In section 3.5 two methods of declaring parameters as outputs in the generated module were presented. Unfortunately, assigning values to an input-parameter is undefined behaviour in C. In the LLVM IR output from the compiler, no assignment to any parameter is performed. The alternative of adapting the clang-compiler to treat name-prefixed input-parameters as outputs were considered, but it would be time-consuming to dig into the clang-libraries as well. By disabling optimization of the IR or declaring the output-parameter as *volatile*, the assignment operation is present in the IR-code. Unfortunately, the assignment in the IR-code is not to a variable but to a local RAM module, generated for the parameter. No assignments to the output exists.

When disabling the optimization-passes in the compiler, some patterns were noticed that could provide useful information. The idea was to look for assignment-information in the LLVM IR-code, which could be used in LegUp to assign the correct values to the output. To show how this information can be used to assign values to outputs, it is best to use a simple example. In listing 3.1, a short C-code is listed. When running the pure-HW flow of LegUp on this code, the human readable format of the LLVM IR-code is output to a file named *designName.ll*.

```
1 void main(int inDataA, int inDataB, volatile int __out_outData) {
2   while (1) {
3     __out_outData = inDataA * inDataB;
4   }
5   return;
6 }
```

Listing 3.1: Simple C-code example for LLVM IR parsing

```
1  define void @main(i32 %inDataA, i32 %inDataB, i32 %__out_outData) #0 {
2    %1 = alloca i32, align 4
3    store volatile i32 %__out_outData, i32* %1, align 4
4    br label %2
5
6  ; <label>:2                                       ; preds = %2, %0
7    %3 = mul nsw i32 %inDataA, %inDataB
8    store volatile i32 %3, i32* %1, align 4
9    br label %2
10                                                   ; No predecessors!
11   ret void
12 }
```

Listing 3.2: LLVM IR code for simple parsing example

Lets analyze the content of this file, shown in listing 3.2. On line 2 the temporary register %1 is created. On line 3, the input parameter declared as volatile, *___out_outData*, is stored to this register. On line 7, the calculated multiplication of the inputs *inDataA* and *inDataB* is stored to a new temporary register, %3. On line 8, the content of register %3 is stored back to register %1. This information can be exploited to create a program that traces stores, back to the original input-parameter. In this example it is easy to see that the storing of the calculated multiplication can be traced back to the output *___out_outData*, but in more complex programs, this tracing might not be that simple. One solution is to create a script that parses through the IR-code and makes these connections. Notice that the parameter that should be an output **needs** to be declared as *volatile*, if not, the first allocation and store operations will be removed by link-time optimization passes. Optimization cannot be disabled, as this leads to temporary registers being used for all parameters rather than parameter-names, causing problems for streaming inputs described in section 3.7.

When LegUp generate signals, they will be named by the convention:

`functionName_labelNumber_registerNumber`.

The example above will then create the signals *main_0_1* from line 2 and *main_2_3* from line 7. As the first line describes an *alloca*-operation, *main_0_1* will actually be implemented as a RAM-module. RAM-modules will be generated for all input-parameters declared as *volatile*. This is good news, as this is needed for the program that trace assignments to outputs.

3.6.1 LLVM IR assignment parser program

A program have been created to parse the LLVM IR generated by the compilation. The code for this program is written in the language C++. The reason for the language choice is merely that this was a familiar language for the writer of this thesis. The size of the program was not thought to be large enough for it to be beneficial to look into another language, given the limited amount of time available. In hindsight, a scripting language like Perl or Python could presumably be preferred for this kind of task. This section will explain in words and pseudo-code how the program works. The full source code of the parser program is included in appendix A.2. The program takes two command-line arguments when called, the name of the input file and the name of the output file. The input file should be the final LLVM IR file generated by LegUp, named *designName.ll*. The output filename can be anything, but the default filename used in LegUp for reading the output-file is *LLVMParsed.log*. The program consist of two parts, the first part handles the reading and parsing of the input file, the second part handles tracing and generating of the output file. The program is created to only care about the *main*-function, as this is the module where it is vital to have multiple output signals. The program can easily be changed by altering the source code, if additional functionality is needed.

A pseudo-code describing the first part of the program is shown in algorithm 3.2. The parser starts by looking for the *main*-function. When in the *main*-function, the program looks for lines containing stores or labels. If a store is found, the source and target register of the store, together with the current label, is stored in separate vectors. If a new label is found, the label is set as the current label.

A pseudo-code describing the second part of the program is shown in algorithm 3.3. A double for-loop is needed to check each target against all other target. The C-example above will store the values shown in table 3.1 in the vectors. By comparing the first target against the second target in the table, it can easily be seen that the second source is stored to the same target as the first source. Notice that this program uses the name-prefix described in section 3.5.1. This method of declaring outputs is therefore required for the parser program to work.

Algorithm 3.2 Input file handling in LLVM IR parser program

Require: inFile and outFile should be passed as arguments

```
1: if inFile.open() then
2:     currentLabel ← 0
3:     inMain ← false
4:     while inFile.getNextLine() ≠ inFile.end() do
5:         if inMain then
6:             if lineStartWith() = "  store" then
7:                 newSource ← sourceRegisterFromLine
8:                 newTarget ← targetRegisterFromLine
9:                 sources.insert(newSource)
10:                targets.insert(newTarget)
11:                labels.insert(currentLabel)
12:            else if lineStartWith() = "; < label >: " then
13:                currentLabel ← labelNumberFromLine
14:            else if lineStartWith() = "}" then
15:                inMain ← false
16:            end if
17:        else if lineStartWith() = "define %type% @main" then
18:            inMain ← true
19:        end if
20:    end while
21: end if
```

sources	targets	labels
___out_outData	1	0
3	1	2

Table 3.1: Vector values after parser run

The program will output the result of the tracing into a file with the name given as parameter to the program. The format of the output is:

```
"sources[i] sources[j] labels[j] labels[i] targets[i]"
```

The output from the above example will then be:

```
"outData 3 2 0 1".
```

This implies that the signal *main_2_3* should be assigned to the output *outData*. The two last values, 0 and 1, are included as they will be used in a trick in section 3.6.2 to simplify the process of assigning signals to output ports.

Algorithm 3.3 Output file handling in LLVM IR parser program

```
1: if outFile.open() then
2:     done ← {}
3:     for i ← 0 to targets.size() do
4:         for j ← 0 to targets.size() do
5:             if targets[i] = targets[j] and i ≠ j and sources[i] ∉ done then
6:                 newSource ← sourceRegisterFromLine
7:                 newTarget ← targetRegisterFromLine
8:                 done.insert(sources[j])
9:                 if sources[i].lineStartWith() = "___out_" then
10:                    parameterName ← sources[i].strip(___out_)
11:                    outFile.print(parameterName)
12:                    outFile.print(sources[j])
13:                    outFile.print(labels[j])
14:                    outFile.print(labels[i])
15:                    outFile.print(targets[i])
16:                    outFile.print("\n")                    ▷ Newline
17:                end if
18:            end if
19:        end for
20:    end for
21: end if
```

Execution of the program is added to the Makefile, *Makefile.common*, just before running the LegUp backend pass that generated Verilog output. The program is called by the line:

```
$(LEVEL)/LlvmParser.run $(NAME).ll LlvmParsed.log
```

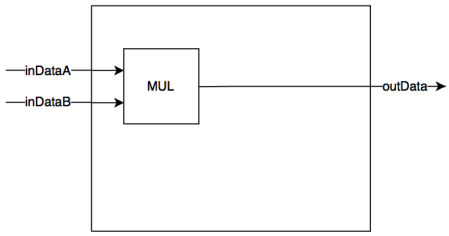
3.6.2 Assigning output signals

As described in section 2.2.2, any RTLSignal that exist in a RTLModule can be found by calling the function *find()*, with the name of the signal passed as a string parameter. A RTLSignal can have multiple drivers and conditions, and the i-th driver or condition can be found by calling *getDriver(i)* and *getCondition(i)*. The initial idea when the LLVM IR parser program was created, was to find each of the signal and connect them to the correct output.

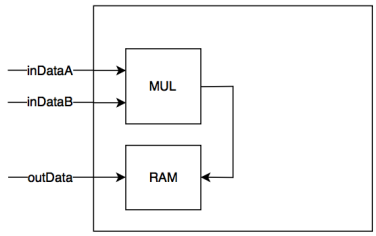
For every parameter to the function, the compiler will allocate a register and store the parameter value to this register. Whenever a store to a parameter is performed, this value will be stored to the first allocated register. In LegUp, the allocated register will be implemented as a RAM module and all stores to the parameter will

be stored to this ram. This information can be exploited to re-assign values stored to this RAM, to the output port instead.

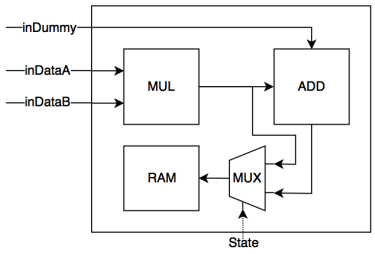
Figure 3.1 tries to illustrate the problem with assigning outputs. In figure 3.1a, the module we would expect from the C-code in listing 3.1 is shown. The two inputs are multiplied together and output to *outData*. In reality, what happens in the generated Verilog is shown in figure 3.1b. In figure 3.1c, the example is extended with an extra ADD module that stores to *outData*. Instead of assigning the calculated values to the output, they are stored in a RAM module. In figure 3.1c, the current state is used to decide which signal is input to the RAM. The solution to this problem is shown in figure 3.1d. By "hijacking" the input signal to the RAM module and assigning it to the output, *outData*, we get the expected functionality.



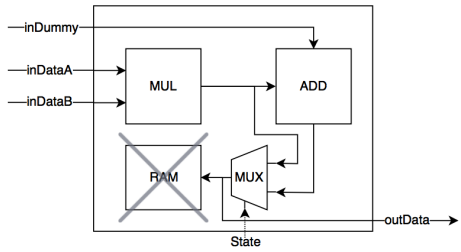
(a) What we want to achieve



(b) What the compiler/Legup thinks we want to achieve



(c) An example with multiple stores to *outData*



(d) The solution

Figure 3.1: Problem with assigning values to output

The RAM module will be named by the same convention as signals, naming the RAM from the example *main_0_1*. The data input-signal of the RAM module is named *ramName_in_a*. This is the signal we want to "hijack". The "hijacking" is performed as described in algorithm 3.4. The name of the output ports are read from the file output from the LLVM IR parser program, together with the name of the corresponding RAM modules. Each driver-condition pair in the input-signal of

the RAM module, is added as a conditional driver to the output port.

Algorithm 3.4 Assigning values to outputs

```
1: for i ← 0 to outputPorts.size() do
2:     outputPort ← find(output signal name)
3:     ramSignal ← find(RAM module inData signal)
4:     for j ← 0 to ramSignal.getNumDrivers() do
5:         outputPort.addCondition(ramSignal.getDriver(j),
                                   ramSignal.getCondition(j))
6:     end for
7: end for
```

3.6.3 Removing local RAMs

As described in section 3.6, each parameter declared as *volatile* will generate a RAM module. After reassigning the stores to the output port, the generated RAM modules are no longer needed. These RAM modules can be removed to save area and reduce power consumption. To make this operation optional, the TCL-parameter *REMOVE_UNUSED_LOCAL_RAMS* was added. By setting this parameter, the local RAM modules will be removed. All local RAMs are stored together with its corresponding function (from the C-code) in a variable, *isLocalFunctionRam*. If the RAM is removed from this variable, it is also remove the generated Verilog. In addition to removing the RAM module, all signals to and from the RAM must be removed as well. Notice that only the RAMs generated by output parameters are removed from the *main*-module. RAM modules can also be generated by arrays and other large data structures, but these will not be removed. This method of removing the RAMs does not remove the states for allocation and stores to the RAMs present in the FSM generated by LegUp.

3.7 Streaming inputs/outputs

For most module designs to be useful and fast, it must be able to continuously take new inputs and generate outputs, without having to start and stop the entire module each time, with all the overhead in time this would require. The way LegUp is designed, functions are used as hardware accelerators, meaning it gets some input, performs some calculations and then outputs the result. The module is then finished and will not run again until next time the accelerated function is called. For this approach to work for an ASIC implementation, a top module would need to be created to assign new inputs and start the module again once it is finished with the last iteration. If the output-value is used in the next run, a feedback loop needs to be added to pass the result back to the new inputs. The concept is shown in figure 3.2.

This solution would be hard to implement and would create extra overhead, both in terms of speed and area of the design. Another solution is to add a while loop inside the *main*-function of the C-code to make the program run continuously. The initial problem with this approach is that to output a value from the function, the *return*-statement is used. Calling *return* will cause the program to terminate, which is not desirable. With the method implemented in section 3.6, it is no longer necessary to call *return* to output a value. This while-loop method can therefore be used with this altered version of LegUp.

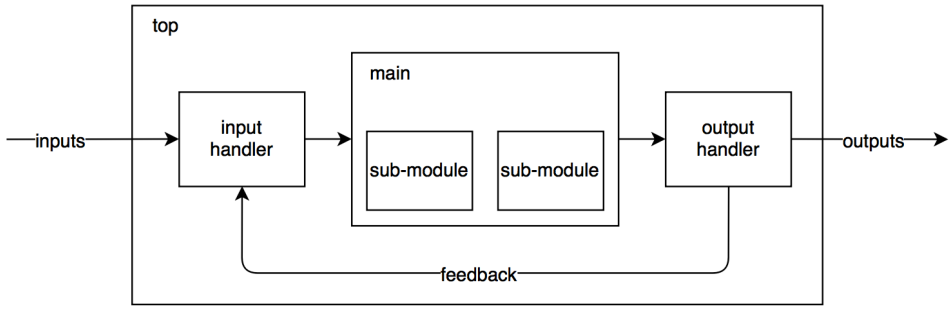


Figure 3.2: Top-level concept for streaming inputs and outputs

With this solution, some new issues arise. First we need a way to stop the module if all calculations are finished. This can easily be handled by adding a input parameter to the *main*-function in the C-program, lets call it *done*, which is used as condition for running the while-loop. This parameter will then correspond to a signal in the Verilog-module that can be used to terminate the module. No alterations to LegUp is performed to resolve this issue, as it can be resolved manually by the user. This signal could potentially be integrated to the Verilog-generation in LegUp, but this would require major alterations to the libraries and the generated data-flow.

Secondly, we need a way to know when an output has valid data. A simple solution here is to generate a valid-flag for each output signal. These flags are created simultaneously with the outputs being connected to the driving signals, as described in 3.6. The signals shall be valid only in the first clock cycle after the output signal has changed. To achieve this, two condition signals have to be created, one when the output is valid, and one when the output is not valid. The valid condition shall be set in any state where the output signal is assigned. This means that the conditions used to set the RAM-module's data input-signal can be used. The not-valid signal should be set in all other states. To generate the valid-signal is straightforward, as the condition-signals for storing values to the RAM is already generated by LegUp. These conditions can be copied from the RAM-signal and added to the output-valid signal as conditional drivers. The not-valid signal is a bit more tricky to generate.

By creating an RTLOp-signal that ANDs together all the valid states, and creating a second RTLOp-signal that NOTs this signal, the desired not-valid signal is created. Figure 3.3 illustrates how the signal is generated. The tricky part arises from the AND-operation only being able to take two operands, making it hard to create code handling special cases of few and odd number of valid states. The source code of how this is handled is included in appendix A.3.

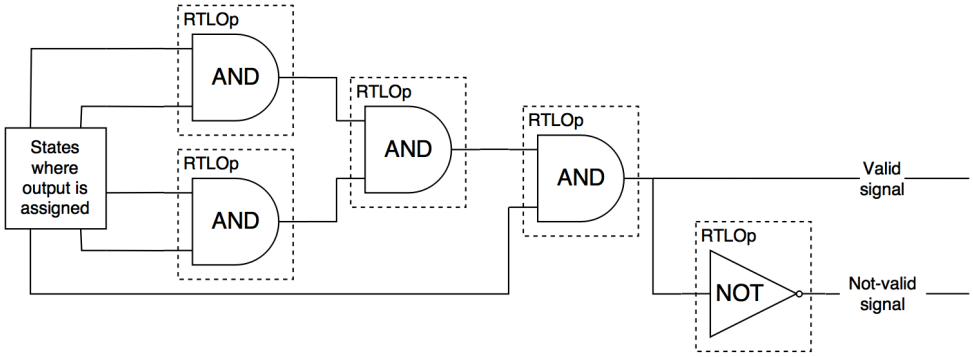


Figure 3.3: Generating not-valid signal.

As each output can be valid at different times, and also multiple times during a loop, a third issue needs to be handled. A way to know when an iteration of the loop is finished must be added. A flag, *iterationFinish*, can be added by setting the parameter *ENCLOSING_LOOP* in the constraint file. The flag will be set each time the loop condition-check is performed (except the first time). If the code is divided into three parts, pre-loop, loop, and post-loop, the implementation of this solution can be simplified a bit. By not allowing any operations, except return, to be executed in the post-loop part, as shown in listing 3.3, the flag only needs to be set in the state preceding the final state of the FSM.

```
1 void main( int done ) {
2   //Pre-loop: Variable setup etc. can be done here.
3   while(done == 0) {
4     //Loop: Functional operations are performed here
5   }
6   //Post-loop: No operations can be done here.
7   return;
8 }
```

Listing 3.3: Sectioning of a program with enclosing while-loop

The *iterationFinish*-flag is set to zero in all other states. This is done by adding an RTLOp-signal that NOTs the condition for setting the flag high. This RTLOp-signal is added as a condition for driving the flag low. The source code of this is listed in appendix A.4.

3.8 Signal sizes

A problem with writing the functional specification in C is that C have no built-in support for bit-sizes. Data types of sizes 1, 8, 16, 32 and 64 bits are defined, but if you want to declare a signal of any other bit-width, like you often do in hardware design, this is not possible. The problem with over-sized signal sizes is that larger circuits will be generated in order to handle the calculation of the expected extra bits. Two solutions were looked upon for solving this issue; bit-packed structs and bit-width attributes. In C it is supported to define a variable in a struct to be any given number of bits. The thought behind this feature is to allow storing multiple small variables in the memory space occupied by the entire struct. In the example shown in listing 3.4 the struct *s* will occupy 12 bytes, as one int equals 4 bytes, while the struct *t* will occupy 3 bytes, as the total defined bit-width is 23 bits.

```
1  struct s {
2    int a;
3    int b;
4    int c;
5  }__attribute__((packed));
6
7  struct t {
8    int a:1;
9    int b:15;
10   int c:7;
11 }__attribute__((packed));
```

Listing 3.4: Struct bit-packing example

There are two issues with this solution; assigning values to a variable in a struct takes much more typing than other variables, even if the struct only contains a single variable, and structs are not supported by the inferred RAMs generated by LegUp. This means that if structs are used in the design, *altsyncram*-modules are generated in place of inferred RAMs.

The other alternative that has been explored is to use a attribute that sets the bit-width of a variable. By adding `__attribute__((bitwidth(N)))` to the end of the variable declaration, where `N` is the number of bits in the signal, the compiler can define the signal size based on this attribute. Implementation of this attribute was suggested as an addition to *llvm-gcc*, the GCC frontend compiler for LLVM, in 2007 [31], but unfortunately it was discarded in 2010. This attribute would allow for defining new data types for each size, and using them as standard data types:

```
1  typedef int __attribute__((bitwidth(2))) int2;
2  typedef int __attribute__((bitwidth(4))) int4;
3  typedef int __attribute__((bitwidth(25))) int25;
4
5  int2 a = 3;
6  int4 b = 8;
7  int25 c = 2502;
```

which could be translated into the following in the LLVM IR:

```
1 @a = global i2 3, align 4
2 @b = global i4 8, align 4
3 @c = global i25 2502, align 4
```

The align value could be lower, depending on the number of bits/bytes allocated to each temporary register. This solution is possible to implement in theory, but it would require changing both the clang compiler frontend for LLVM and the LegUp backend pass. Due to the estimated amount of time these alterations would require, this issue has not been resolved. For this thesis it is not vital that exact signal sizes can be set, as long as the same sizes are used in both C and Verilog code to get a fair comparison.

3.9 Testbench generation

The original version of LegUp generate a basic testbench shell, but this is very static. It is also incorporated into the same file as the RTL-design, making it impractical to use in the desired framework. The generated testbench consists of a testbench module, *main_tb*, which instantiate the *top*-module and sets *reset*, *start* and *waitrequest* flags. The input and output signals in the *top*-module does not contain custom signals from the *main*-module, and in an ASIC implementation we are not interested in the memory controller and additional modules instantiated in the *top*-module. The implemented solution is to instantiate the *main*-module in the testbench-module and add each input or output by iterating over the ports in the *main*-module. By setting the TCL-parameter *SEPARATE_TB_FILE*, the testbench will be output to a separate file from the RTL-design. The source code of how the testbench-generation is performed is listed in appendix A.5.

As the testbench does not come with any form of testcases or applied signals, the testbench generator is extended to input Verilog code from a file specified by the TCL-parameter *TB_TESTCASE_FILE*. This allows the user to specify testcases in this file, that will be automatically inserted into the testbench file. The code will be placed inside the testbench-module, but not inside any procedural blocks. This allows the user to add the preferred procedural block in the specified testcase file.

An example testcase file can then be:

```
always @(iterationFinish) begin
  if (iterationFinish == 1) begin
    $display("At t=%t, Loop iteration finished", $time);
  end
end
```

or

```
initial begin
    inData <= 100;
    @(posedge clk)
    inData <= 0;
    @(posedge clk);
    $display("At t=%t, outData=%d", $time, outData);
end
```

This insertion of testcases, enables the script to automatically run HLS and thereafter run simulation, using the generated design and testbench.

3.10 Coding constraints

Due to the described alterations to the LegUp libraries, some guidelines need to be followed when writing the functional specification, to ensure correct output. The following subsections will describe these guidelines.

3.10.1 Structs

To support structs, byte-enable must be supported by the RAM or ROM module used to store the data. The RAM and ROM modules inferred by LegUp does not support byte-enable, resulting in struct support not being present when writing the functional specification. If structs are used in the code, LegUp will use altsyncram-modules instead of inferring RAMs. The altsyncram-module is not supported by the tool-flow used at Nordic Semiconductor, and inferred RAM-modules must therefore be used.

3.10.2 Pointers

Pointers are used to reference an object in memory, opposed to passing a copy of the actual object between function. This reduces both Central Processing Unit (CPU)-time and memory-space, as objects does not need to be copied every time it is used, and makes it possible to alter a memory object directly without implicit load and store operations. As the memory controller used to pass data between different modules are unwanted in an ASIC implementation, support for pointers are limited to use inside the function where the pointer is declared. This limitation is a big drawback with this altered version of LegUp.

3.10.3 Arrays

Arrays can be used in the programs to some extent, but if the arrays get too large, or too many arrays are instantiated, LegUp will implement these as RAMs inside the global memory controller. The reason for this is that arrays in C basically is a pointer to the array type. When the point-to analysis cannot determine that a single function uses the array, it is automatically implemented as a global RAM.

3.10.4 Inputs and outputs

The implemented method of adding inputs and outputs to the module is not ideal. The need for using name-prefixes when writing the C-code puts more of the work on the coder, and draws the input further away from the standard ANSI-C that was intended as the input language. If using a while loop for supporting streaming inputs and outputs, it is not possible to do calculations after the loop, as this will break generation of the *iterationFinish*-flag. Not being able to specify signal sizes also limits what designs can be created using the tool. All these limitations has to be taken into consideration when writing the functional specification, to make sure the design is supported by the tool-flow and framework.

Chapter 4
Tool-flow example

This chapter will give a detailed description of the information- and tool-flow, using the adapted version of LegUp and the other tools used in the creation of a framework. A simple C-code example will be used, and the flow and generated information will be described all the way through HLS, simulation, synthesis, layout and power analysis. Listing 4.1 shows a simple C-code with two functions, *main* and *squared*. The *main*-function will be the top-level function, taking three input parameters; *done*, *inData* and *___out_outData*. The *main*-function contains a while loop, which will run as long as the input parameter *done* is set to zero. Inside the loop the function *squared* is called with the argument from *inData* and the return value from *squared* is assigned to the input argument *___out_outData*. Readers familiar with C programming might think that it is a weird thing to be assigning values to a non-pointer parameter, but as described in section 3.6, this is implemented as a way to get multiple outputs to the generated module. Notice also the *volatile* keyword in the declaration of the *___out_outData* parameter, as this is a necessary part of generating outputs.

```
1  int squared (int base) {
2    return base * base;
3  }
4
5  int main (int done, int inData, volatile int __out_outData) {
6    while(done == 0) {
7      __out_outData = squared(inData);
8    }
9    return 0;
10 }
```

Listing 4.1: Simple C-code example

4.1 HLS with LegUp

The HLS-tool is the only part of the tool-flow located on another computer than the rest. This is not an issue when running a single design through the tool-flow, as the

code can simply be copy-pasted to the destination. As suggested in [13], SSH and SCP can also be used for copying files and executing commands on remote machines.

4.1.1 Constraint files

LegUp use constraint files to set constraints and settings for the HLS. The default values for necessary constraints is set in the default constraint-file located in *~/legup4-0/example/legup.tcl*. The default constraints can be overridden by adding a local constraint file. The filename of the local constraint file must be specified in the Makefile for the constraints to take effect. This is done by adding the line "`LOCAL_CONFIG = -legup-config=config.tcl`" to the Makefile, where *config.tcl* is the filename of the local constraint file. The example code is run with the constraints listed as *required* in section 2.2.3. All other constraints are left to default values.

4.1.2 Makefile

A local Makefile is required to compile the project. The minimal local Makefile of LegUp contains the following three lines:

```
NAME=name
LEVEL = ..
include $(LEVEL)/Makefile.common
```

The Variable *NAME* is the name of the design. This variable will be used to name the output-files of the design. The Variable *LEVEL* indicates the number of directory-levels down to the design from the directory where the common Makefile is located. 1 level equals "`..`", 2 levels equal "`../..`" and so on, just like in a standard Linux shell. Other parameters and flags like *NO_OPT* and *NO_INLINE*, can be added to the Makefile to disable optimization and avoid inlining of functions in the compiler. In some cases, especially with simple test-programs, these two flags are necessary to prevent the compiler from optimizing away the whole program.

4.1.3 Compilation

The C-code is compiled into LLVM IR using the clang frontend for the LLVM compilation framework. The initial result before any Link-Time-Optimization (LTO) is performed, is shown in listing 4.2. Readers familiar with assembly code might recognize some of the operations, like *alloca*, *load*, *store*, *mul* and *icmp*. Each *define*-block corresponds to one function from the C-code. The *main*-function, which contains a while-loop, is split into multiple labels. The first section refers to memory-allocation and storing of the input-parameters. Temporary registers, declared on the form %1, %2 .. %N, are inserted where needed.

The section from the line "*; <label>:4*" is the checking of the condition of the while loop. Further, *label 7* is operations inside the while loop, and *label 10* is operations after the loop has exited.

```
1  define i32 @squared(i32 %base) #0 {
2    %1 = alloca i32, align 4
3    store i32 %base, i32* %1, align 4
4    %2 = load i32* %1, align 4
5    %3 = load i32* %1, align 4
6    %4 = mul nsw i32 %2, %3
7    ret i32 %4
8  }
9
10 ; Function Attrs: noinline nounwind
11 define i32 @main(i32 %done, i32 %inData, i32 %__out_outData) #0 {
12   %1 = alloca i32, align 4
13   %2 = alloca i32, align 4
14   %3 = alloca i32, align 4
15   store i32 %done, i32* %1, align 4
16   store i32 %inData, i32* %2, align 4
17   store volatile i32 %__out_outData, i32* %3, align 4
18   br label %4
19
20 ; <label>:4                                ; preds = %7, %0
21   %5 = load i32* %1, align 4
22   %6 = icmp eq i32 %5, 0
23   br i1 %6, label %7, label %10
24
25 ; <label>:7                                ; preds = %4
26   %8 = load i32* %2, align 4
27   %9 = call i32 @squared(i32 %8) #1
28   store volatile i32 %9, i32* %3, align 4
29   br label %4
30
31 ; <label>:10                               ; preds = %4
32   ret i32 0
33 }
```

Listing 4.2: LLVM IR before LTO

4.1.4 Link-time optimizations

By default, some optimization passes are run on the IR to remove unnecessary operations and reduce the number of registers needed. All available passes are described in [22], but the ones that is run by default is *mem2reg*, *instcombine*, *loops*, *loop-simplify*, *basicaa*, *indvars*, *loop-pipeline*, *internalize*, and *globaldce*. These passes analyses, simplifies, and removes operations. The post-LTO-code is listed in listing 4.3. Notice that the only stores left in the code is the ones connected to the input parameter declared as *volatile*. The keyword *volatile*, which is defined as *likely to change suddenly and unexpectedly*, tells the compiler to avoid performing optimizations on this object, as it might change in an unexpected way. To avoid the undefined behavior of assigning values to an input-parameter to be removed by optimization passes, this keyword is necessary.

```
define internal i32 @squared(i32 %base) #0 {
  %1 = mul nsw i32 %base, %base
  ret i32 %1
}

; Function Attrs: noinline nounwind
define i32 @main(i32 %done, i32 %inData, i32 %__out_outData) #0 {
  %1 = alloca i32, align 4
  store volatile i32 %__out_outData, i32* %1, align 4
  br label %2

; <label>:2                                 ; preds = %4, %0
  %3 = icmp eq i32 %done, 0
  br i1 %3, label %4, label %6

; <label>:4                                 ; preds = %2
  %5 = call i32 @squared(i32 %inData) #1
  store volatile i32 %5, i32* %1, align 4
  br label %2

; <label>:6                                 ; preds = %2
  ret i32 0
}
```

Listing 4.3: LLVM IR after LTO

Also notice that the storing and loading of the other input parameters have been swapped by referencing the input parameter name directly. For instance the four lines:

```
%2 = alloca i32, align 4
store i32 %inData, i32* %2, align 4
%8 = load i32* %2, align 4
%9 = call i32 @squared(i32 %8) #1
```

has been transformed into the single line:

```
%5 = call i32 @squared(i32 %inData) #1
```

4.1.5 Verilog generation

The LegUp backend pass for LLVM is run on the IR to generate Verilog. LegUp performs allocation, scheduling and build RTL-models of the functionality, based on given constraints. These RTL-models are printed to a file as Verilog HDL. Listing 4.4 shows the declaration of the *main*-module, parameters representing states, port- and internal signal-declarations, and instantiation of submodules from the output Verilog.

```
module main (
  clk,
  clk2x,
  clk1x_follower,
  reset,
  start,
  finish,
  memory_controller_waitrequest,
  return_val,
  arg_done,
  arg_inData,
  arg_outData,
  arg_outData_valid,
  iterationFinish
);

parameter [3:0] LEGUP_0 = 4'd0;
parameter [3:0] LEGUP_F_main_BB__0_1 = 4'd1;
parameter [3:0] LEGUP_F_main_BB__0_2 = 4'd2;
parameter [3:0] LEGUP_F_main_BB__2_3 = 4'd3;
parameter [3:0] LEGUP_F_main_BB__4_4 = 4'd4;
parameter [3:0] LEGUP_F_main_BB__4_6 = 4'd6;
parameter [3:0] LEGUP_F_main_BB__4_7 = 4'd7;
parameter [3:0] LEGUP_F_main_BB__6_8 = 4'd8;
parameter [3:0] LEGUP_function_call_5 = 4'd5;

input  clk;
input  reset;
input  start;
output reg  finish;
input  memory_controller_waitrequest;
output reg [31:0] return_val;
input [31:0] arg_done;
input [31:0] arg_inData;
output reg [31:0] arg_outData;
output reg  arg_outData_valid;
output reg  iterationFinish;
reg [3:0] cur_state;
reg [3:0] next_state;

squared squared (
  .memory_controller_waitrequest (memory_controller_waitrequest),
  .clk (clk),
  .clk2x (clk2x),
  .clk1x_follower (clk1x_follower),
  .reset (reset),
  .start (squared_start),
  .finish (squared_finish),
  .return_val (squared_return_val),
  .arg_base (squared_arg_base)
);
```

Listing 4.4: Verilog module, port, signal and parameter declaration, and sub-module instantiation

LegUp generates a FSM that perform calculations and generate outputs. The generated FSM-controller for the example C-code is listed in listing 4.5, and the state diagram of the FSM is shown in figure 4.1.

```
always @(posedge clk) begin
if (reset == 1'b1)
  cur_state <= LEGUP_0;
else if (memory_controller_waitrequest == 1'd1)
  cur_state <= cur_state;
else
  cur_state <= next_state;
end

always @(*)
begin
next_state = cur_state;
case(cur_state)  // synthesis parallel_case
LEGUP_0:
  if ((start == 1'd1))
    next_state = LEGUP_F_main_BB__0_1;
LEGUP_F_main_BB__0_1:
    next_state = LEGUP_F_main_BB__0_2;
LEGUP_F_main_BB__0_2:
    next_state = LEGUP_F_main_BB__2_3;
LEGUP_F_main_BB__2_3:
  if ((main_2_3 == 1'd1))
    next_state = LEGUP_F_main_BB__4_4;
  else if ((main_2_3 == 1'd0))
    next_state = LEGUP_F_main_BB__6_8;
LEGUP_F_main_BB__4_4:
    next_state = LEGUP_function_call_5;
LEGUP_F_main_BB__4_6:
    next_state = LEGUP_F_main_BB__4_7;
LEGUP_F_main_BB__4_7:
    next_state = LEGUP_F_main_BB__2_3;
LEGUP_F_main_BB__6_8:
    next_state = LEGUP_0;
LEGUP_function_call_5:
  if ((squared_finish_final == 1'd1))
    next_state = LEGUP_F_main_BB__4_6;
default:
  next_state = cur_state;
endcase
end
%
```

Listing 4.5: Verilog FSM

By looking at the state diagram, some parts from the C-code can be recognized. *0* is the initial state and the two following states is allocating and storing of the volatile input parameter. *3* is where the condition checking for the while loop is performed. The states *4-7* are the states inside the while loop, while *8* is the exit-state, signalizing the completion of the program.

Some signal assignments are shown in listing 4.6. Each assignment is printed with the corresponding LLVM operation commented above, to increase readability of the code. The later assignment of output signals are not generated directly from an LLVM operation and does thus not have this operation printed along the assignment. These signals are generated by the alterations made to LegUp, described in chapter 3.

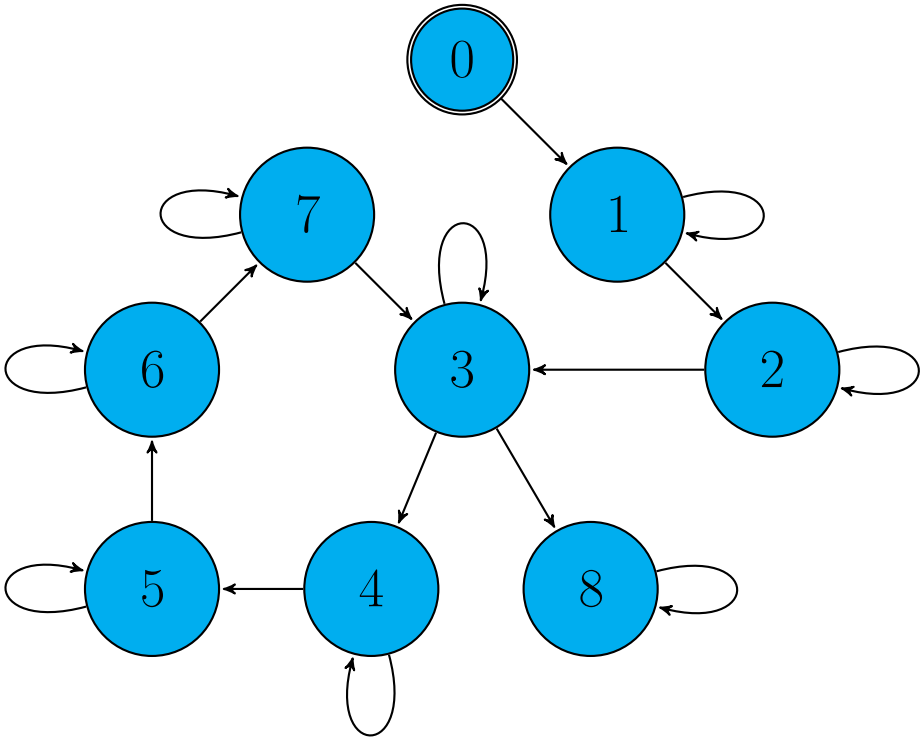


Figure 4.1: State diagram of generated FSM

```
always @(*) begin
  /* main: %2*/
  /*   %3 = icmp eq i32 %done, 0*/
    main_2_3 = (arg_done == 32'd0);
end
always @(posedge clk) begin
  /* main: %4*/
  /*   %5 = call i32 @squared(i32 %inData) #1*/
  if ((cur_state == LEGUP_F_main_BB__4_4)) begin
    squared_arg_base <= arg_inData;
  end
end
always @(posedge clk) begin
  if ((cur_state == LEGUP_F_main_BB__4_6)) begin
    arg_outData <= main_4_5_reg;
  end
end
always @(posedge clk) begin
  if ((cur_state == LEGUP_F_main_BB__4_6)) begin
    arg_outData_valid <= 1'd1;
  end
  if (~((cur_state == LEGUP_F_main_BB__4_6))) begin
    arg_outData_valid <= 1'd0;
  end
end
always @(posedge clk) begin
  if ((cur_state == LEGUP_F_main_BB__4_7)) begin
    iterationFinish <= 1'd1;
  end
  if (~((cur_state == LEGUP_F_main_BB__4_7))) begin
    iterationFinish <= 1'd0;
  end
end
```

Listing 4.6: Verilog FSM

4.2 Simulation

4.2.1 Simulation libraries

LegUp generate a local RAM-module for each output-parameter to the *main*-module, as these parameters needs to be declared as volatile. In the declaration of the modules *ram_dual_port* and *rom_dual_port*, a conversion function from an Altera library is used to convert *.mif* files to a format readable by Modelsim. Initially this library-file needed to be included in the design-filelist for the simulation to run successfully. However, a patch fixing a Xilinx related bug in LegUp [20] resolved this issue, as Xilinx devices support raw memory instantiation files.

4.2.2 Running simulation

The simulation is executed by running the script *RUN_ALL* located in the *sim/run*-directory. The GUI of ModelSim for Questa can be brought up by passing the

argument `-g` to the script, otherwise the simulation will be run in batch mode in the terminal.

To simulate the design, a suitable testbench must be provided. Since LegUp generate a testbench shell, it is only required to add the desired testcases that will be applied to the circuit. Since this is a simple example, only a few testcases are provided to ensure the design works as expected.

```
initial begin
    arg_done <= 32'd0;
    arg_inData <= 32'd20;

    @(negedge reset);
    start <= 1;
    @(negedge clk);
    start <= 0;

    @(posedge iterationFinish);
    arg_inData <= 32'd100;

    @(posedge iterationFinish);
    arg_done <= 32'd1;
end
```

Listing 4.7: Testcases for the example testbench

Listing 4.7 shows the testcases applied to the circuit. Initial values of the inputs *done* and *inData* is set, and the testbench waits for a falling edge of the reset signal, set by the automatic generated testbench shell, before setting the *start*-flag high for one clock cycle. At the following rising edge of the *iterationFinished*-flag, a new value is assigned to the input *inData*. At the second rising edge of the *iterationFinished*-flag, the input *done* is set to one, indicating the end of this run. A waveform from the simulation is shown in figure 4.2. The signals between the blue lines are from the *main*-module, while the signals between the red lines are from the *squared* sub-module. From the waveform the expected behavior of the circuit is observed. When the *start*-flag is set, the FSM starts and the sub-module *squared* starts to calculate the output value. When *cur_state* of the *main*-module equals 7, the calculated value is assigned to the output *outData*, and the *valid*-flag for this output is set. The value on the output is correctly calculated for the given testcases. When *cur_state* equals 3, the iterationFinish-flag is set, as this is the state where the condition for the while loop is evaluated, i.e. one iteration of the loop is complete. Notice that the *iteartionFinish*-flag is not set in the first occurrence of $cur\_state = 3$, as this is the first encounter of the loop and not a finished iteration. When the input *done* is set to 1 after the second *iterationFinish*-flag, we observe that the loop terminates and the *finish*-flag is set, indicating the end of the run.

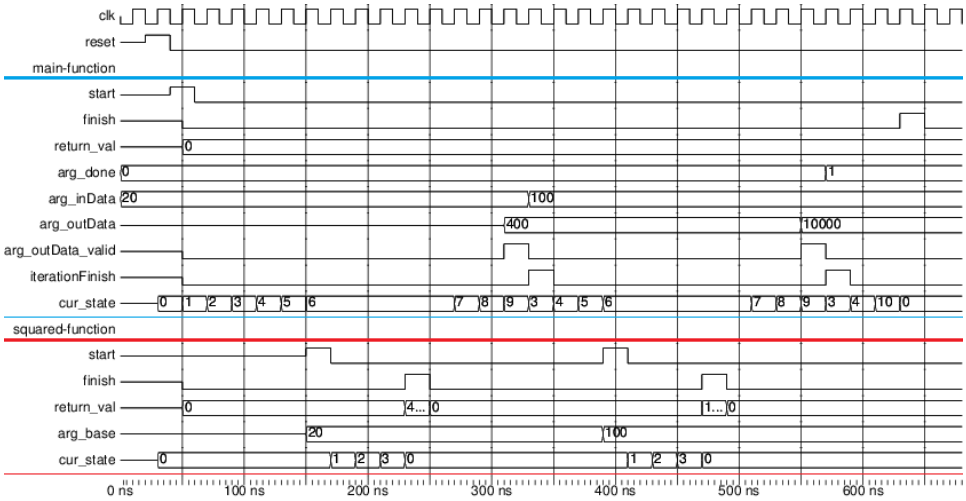


Figure 4.2: Simulation waveform of example design

4.3 Synthesis

The design is synthesized using a 180nm library. After synthesis, the circuitry of the design can be viewed in a GUI tool by running the command "`make open_mapped`" and consecutively calling "`start_gui`" in the *dc_shell* that is opened. The output from LegUp is usually too large to understand, reducing the usefulness of this GUI-tool. In figure 4.3 the top-module view from the tool is shown. The area and frequency reports from the synthesis is presented in table 4.1.

Parameter	**Value**
Combinational Area	1.453853.614678
Noncombinational Area	21015.456253
Buf/Inv Area	1766.318423
Total Buffer Area	39.92
Total Inverter Area	1726.40
Macro/Black Box area	0.000000
Net Interconnect area	42732.128702
Total cell area	44869.070931
Total area	87601.199633
Critical Path Length	27.43 ns
Maximum Frequency	36.46 MHz

Table 4.1: Tool-flow example synthesis results

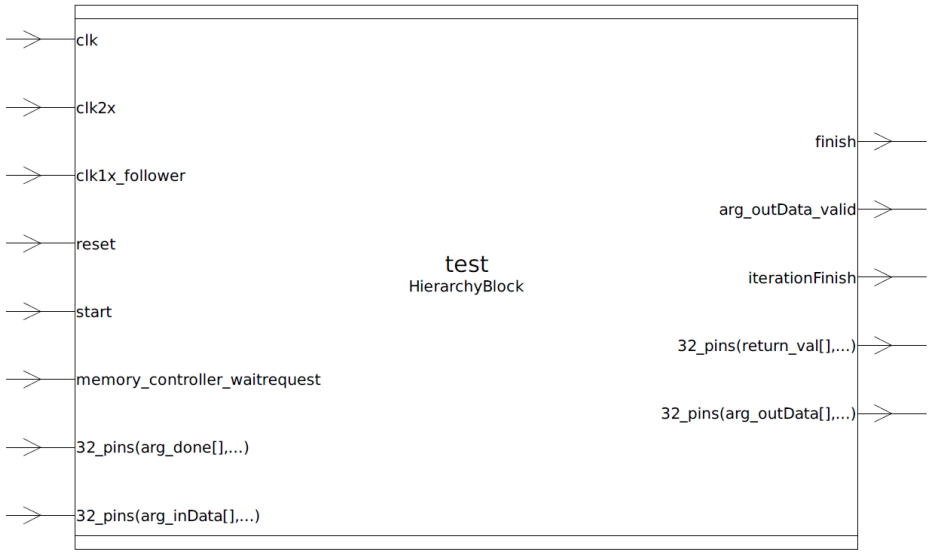


Figure 4.3: Top-level module generated by synthesis

To put the area of the synthesized design in perspective, the library defines a two-port NAND gate to be 9.9792 area units. The number of NAND2-equivalent gates for the design is then 8757 gates. This is a large amount for such a simple design, but the area overhead percentage of the design will shrink with larger designs. The area reported by synthesis is just an estimate, as layout can add optimizations to the design.

4.4 Layout

Layout is performed using the netlist from synthesis. After layout, it is possible to view the layout of the chip in a GUI-tool by opening the *icc_shell* and calling "`start_gui`". The image in figure 4.4 shows the actual layout of the chip. This image is included only for illustration, details are not important. The vertical yellow lines is power distribution to the teal horizontal lines, each purple box is a library cell, and the teal squares along the edges, marked *I* or *O*, are input and output ports of the chip. Layout generates the final reports of area and critical path length. The reported values are listed in table 4.2. Notice that the total area is reduced compared to the synthesis reports, while the critical path has increased a bit, leading to a decreased maximum frequency. The parameters *Net XLength* and *Net YLength* is the physical dimensions of the chip, given in nm. This corresponds to a physical area of 1102 μm^2.

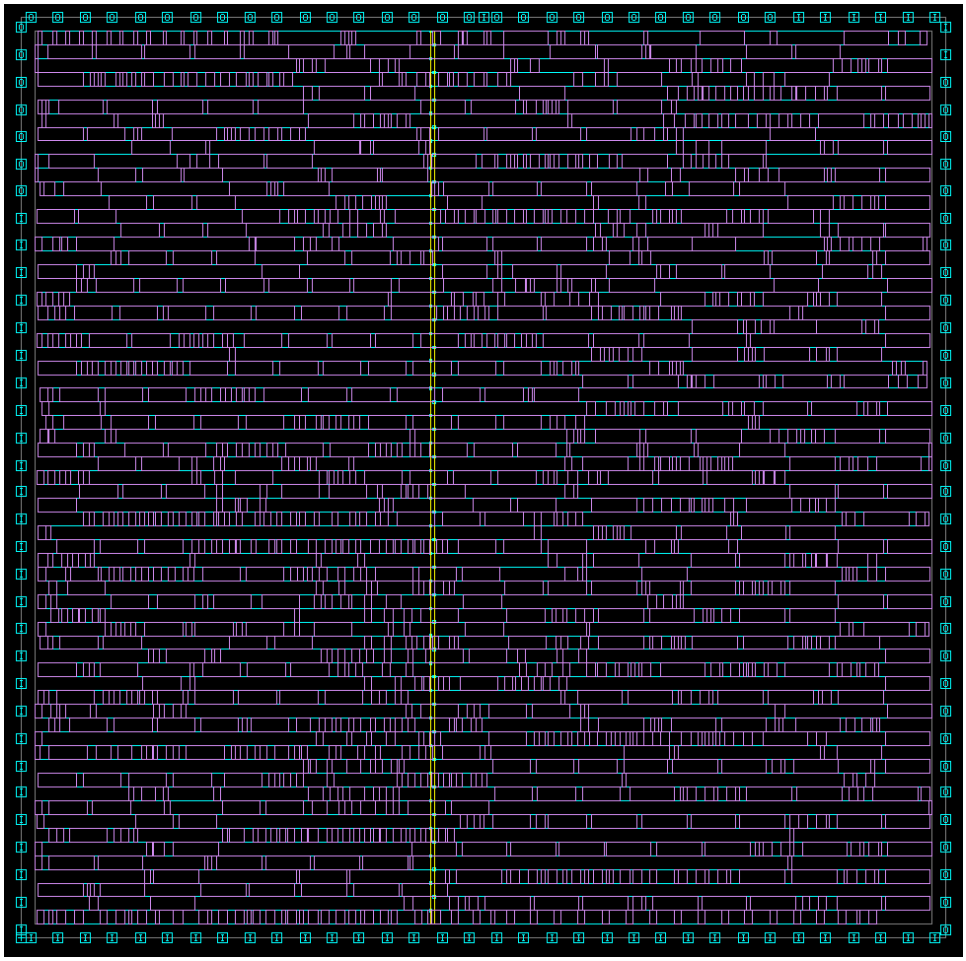


Figure 4.4: Chip-layout of example design

4.5 Power analysis

Power analysis use switching data from the VCD-file generated under simulation to estimate the power consumption. There are five different power scenarios defined in the tool-flow, each generating a result for each power-parameter. The reporting from power analysis is shown in table 4.3. In this simple example, all power scenarios report the same power consumption. In a larger design, these values will vary. With a total estimated power consumption of 179.6μW, not much power is consumed in this chip, but again it is a small design.

Parameter	**Value**
Combinational Area	24698.520289
Noncombinational Area	18075.657349
Buf/Inv Area	2577.960037
Total Buffer Area	844.91
Total Inverter Area	1733.05
Macro/Black Box Area	0.000000
Net Area	0.000000
Net XLength	32867.52
Net YLength	33528.53
Total cell area	42774.177638
Total area	42774.177638
Net Length	66396.05
Critical Path Length	28.51 ns
Maximum Frequency	35.08 MHz

Table 4.2: Tool-flow example layout results

		Power scenario				
Parameter		**ctrl0**	**ctrl1**	**ctrl2**	**ctrl3**	**inactive**
Net Switching	[μW]	27,33	27,33	27,33	27,33	27,33
Cell Internal	[μW]	152,2	152,2	152,2	152,2	152,2
Cell Leakage	[nW]	31,58	31,58	31,58	31,58	31,58
Total	[μW]	179,6	179,6	179,6	179,6	179,6

Table 4.3: Tool-flow example power analysis results

Chapter 5
Creating the framework

This chapter will describe the process of creating a framework for architectural exploration of digital hardware. A few tools that have been implemented to generalize the framework, and allow easy usage for other projects, will also be presented.

5.1 Create new project

To create a new project, the directory hierarchy needs to be copied from a source to the destination, and directories and filenames need to be altered to match the design name. Filelist and setting files also needs to be altered to include the correct design file names. This process is automated into a bash script, *CreateNewProject.sh*. To run the script, the directory *_source*, containing the source project, must be present in the directory where the script is run. The full source code of the script is listed in appendix A.6.

The directory- and file-tree of the framework is shown in figure 5.1. Directories are colored cyan, executables and scripts are colored teal, and other design and constraint files are colored violet. File comment or description are in black. Each file is described in the comment on the right side. The script manages the whole process of copying and renaming files, and replacing the correct strings in the setting files and scripts. When run, the script asks for the name of the new project, and replaces any occurrences of the word *designname* in the *_source*-directory with the given name. The user does not need to change any of the scripts or Makefiles, only update design specific files as described in section 5.3.

5.2 Framework-script

To automate the process of generating multiple design in LegUp and running of the tool-flow on the generated designs, a script is created. LegUp is running on a VirtualBox image and not on the same servers where the rest of the tool-flow are

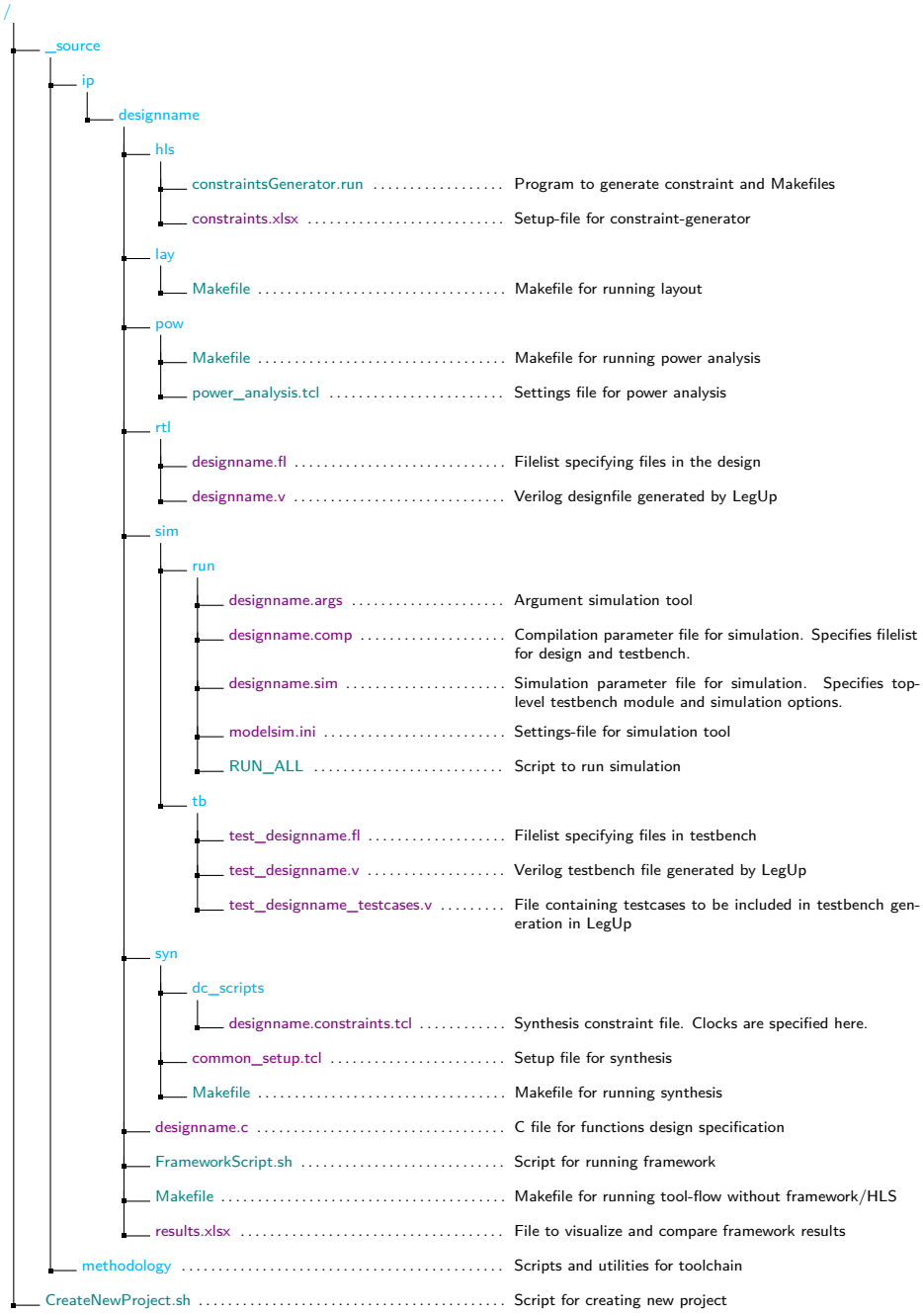


Figure 5.1: Directory and file-tree of the framework

located. This means that some files and commands must be transferred between different machines. In [13], a possible solution using SSH and SCP was proposed. The script is built on this method, but first some additional preparations needs to be made. Since the VirtualBox guest is running on a local computer, a port forwarding rule has to be added to allow connections to port 22 of the guest from the Linux servers of Nordic Semiconductor. The connection has to go through the host computer, as the VirtualBox guest does not have any direct connection to the network. The setting can either be set using the GUI of VirtualBox, or by running the following command from a command line:

```
VBoxManage modifyvm myserver --natpf1 "ssh,tcp,,3022,,22"
```

Here *myserver* is the name of the VirtualBox VM and should be replaced with the name used when the LegUp image was added in VirtualBox. When this setting is added, it is possible to establish a connection over SSH from the Linux server directly to LegUp by connecting to the port 3022 and the local IP-address of the computer running VirtualBox.

The standard SSH and SCP packages on Linux systems does not support passing the password as an argument to the command. To avoid the need to enter username and password each time a file is transferred using SCP, or a command is executed using SSH, it is necessary to setup key-based authentication. This can be done manually, but the framework-script can also do this setup automatically if you pass the flag *-s*. What the script does, is to generate a RSA key-file for the current user with *ssh-keygen* and copy the generated file to LegUp using *ssh-copy-id*. The password is passed to *ssh-copy-id* using *spawn*, *expect* and *send* commands.

The script has seven main tasks:

1. Generate constraint and Makefiles
2. Run HLS-tool to generate Verilog design and testbench files
3. Run simulation
4. Run synthesis
5. Run layout
6. Run power analysis
7. Collect relevant parameters from reports and generate readable result files.

The five tasks in the middle are mostly transferring of files to and from LegUp, and running make-commands. These steps use the tool-flow described in section 2.6. The first and last task are a bit more comprehensive, and is described in the following two subsections. When the script is finished generating and running the tool-flow on one design, the directories *rtl*, *sim*, *syn*, *lay*, and *pow* is copied into a directory named with the designnumber under the *hls*-directory. The script is written for the bash-shell, and the full code of the script is listed in appendix A.7.

5.2.1 Constraint generating

In order to run HLS with a variety of different constraints and settings, one constraint and one Makefile need to be generated for each run. To automate this process, an Excel document, *constraints.xlsx*, has been created. Sheet 2 of this document, shown in figure 5.2, contains the setup of the constraints. Here the user can select which constraints should be randomized and also set if the constraint should have a specific value. Some values are required, and must have the value specified. If a parameter is not needed, the user can specify that the default value of the parameter should be kept. Sheet 1 of the Excel file contains a Comma-Separated Values (CSV) format of the constraint settings. The format of one CSV-string is:

```
parameterName,value,required,random,parameter,makefile,keepDefault
```

Only the fields relevant for the parameter will be printed to the CSV, for instance the parameter *CASE_FSM* will print the line "`CASE_FSM,random,parameter`", as this is the relevant fields for this parameter given the setup of the spreadsheet. Similar, the parameter *NO_OPT* will print the line "`NO_OPT,1,makefile`", since this is a Makefile-parameter defined to have the value 1. The CSV format is copied from the Excel-file to a CSV file by using the headless tool *convert-to* in libreoffice, with the command "`libreoffice --headless --convert-to csv`". The CSV-file can then be read by the program generating constraint- and Makefiles.

The program *constraintGenerator.run* takes the filename of the CSV-file, the level parameter for the Makefile, and the designname as inputs, and returns the number of generated constraint files. This number is used in the framework-script to run the framework the correct number of times. This program is also written in the language C++, with reasoning similar to the one explained in section 3.6.1. By parsing through the above described CSV-file, the program generate one constraint file and accompanying Makefile for each variation of the randomized constraint-parameters. Currently only parameters taking a 1-bit binary value is supported by the constraint generator program. This gives a total of 2^N designs, where N is the number of randomized parameters. The constraints is set using a binary counter, meaning that for 3 randomized parameters, the first constraint file will have the

	A	B	C	D	E	F	G
1	**Constraint name**	**Category**	**Parameter**	**Required**	**Random**	**Value (will be ignored if *Random* or *Keep default* is ticked)**	**Keep default**
2	ASIC_IMPLEMENTATION	HLS	x	x		1	
3	DIVIDER_MODULE	HLS	x	x		generic	
4	EXPLICIT_LPM_MULTS	HLS	x	x		0	
5	INFERRED_RAM_FORMAT	HLS	x	x		xilinx	
6	INFERRED_RAMS	HLS	x	x		1	
7	LOCAL_RAMS	HLS	x	x		1	
8	REMOVE_UNUSED_LOCAL_RAMS	HLS	x	x		1	
9	SEPARATE_TB_FILE	HLS	x	x		1	
10	VSIM_NO_ASSERT	HLS	x	x		1	
11	CASE_FSM	HLS	x		x	0	
12	CASEX	HLS	x			0	
13	CLOCK_PERIOD	HLS	x			0	x
14	DISABLE_REG_SHARING	HLS	x			0	
15	DONT_CHAIN_GET_ELEM_PTR	HLS	x			0	
16	DUAL_PORT_BINDING	HLS	x		x		
17	ENABLE_PATTERN_SHARING	HLS	x		x		
18	ENCLOSING_WHILE_LOOP	HLS	x			1	
19	INCREMENTAL_SDC	HLS	x			0	
20	KEEP_SIGNALS_WITH_NO_FANOUT	HLS	x			0	
21	LLVM_PROFILE	HLS	x			0	x
22	loop_pipeline	HLS				0	x
23	MB_MAX_BACK_PASSES	HLS	x			0	x
24	MB_MINIMIZE_HW	HLS	x		x		
25	MODULO_SCHEDULER	HLS	x			0	
26	MULTIPLIER_NO_CHAIN	HLS	x			0	x
27	MULTIPUMPING	HLS	x			0	x
28	NO_INLINE	Makefile				1	
29	NO_LOOP_PIPELINING	HLS	x			0	
30	NO_OPT	Makefile				1	
31	NO_ROMS	HLS	x			0	
32	PATTERN_SHARE_ADD	HLS	x			0	x
33	PATTERN_SHARE_BITOPS	HLS	x			0	x
34	PATTERN_SHARE_SHIFT	HLS	x			0	x
35	PATTERN_SHARE_SUB	HLS	x			0	x
36	PIPELINE_ALL	HLS	x		x		
37	PIPELINE_RESOURCE_SHARING	HLS	x			0	x
38	PIPELINE_SAVE_REG	HLS	x			0	x
39	PS_BIT_DIFF_THRESHOLD	HLS	x			0	x
40	PS_MAX_SIZE	HLS	x			0	x
41	PS_MIN_SIZE	HLS	x			0	x
42	PS_MIN_WIDTH	HLS	x			0	x
43	RESTRICT_TO_MAXDSP	HLS	x			0	x
44	SDC_BACKTRACKING_PRIORITY	HLS	x			0	x
45	SDC_MULTIPUMP	HLS	x			0	x
46	SDC_NO_CHAINING	HLS	x		x		
47	SDC_ONLY_CHAIN_CRITICAL	HLS	x			0	x
48	SDC_PRIORITY	HLS	x			0	x
49	SEPARATE_TB_FILENAME	HLS	x			test_designname.v	
50	set_combine_basicblock	HLS				0	x
51	set_device_specs	HLS				0	x
52	set_memory_local	HLS				0	x
53	set_operation_attributes	HLS				0	x
54	set_operation_latency	HLS				0	x
55	set_operation_sharing	HLS				0	x
56	set_resource_constraint	HLS				0	x
57	TB_TESTCASE_FILE	HLS	x	x		test_designname_testcases.v	
58	UNROLL	Makefile				0	x

Figure 5.2: Setup of constraint file generation in Excel spreadsheet

values 0,0,0, the second file 0,0,1, the third file 0,1,0, and so on. The generated constraint and Makefiles are output to the directories *constraintfiles* and *makefiles* under the *hls*-directory. The full code of the program generating constraint and Makefiles are listed in appendix A.8.

5.2.2 Report generating

As the framework-script can be used to generate a large amount of designs, it is important to easily be able to collect the relevant data from all the generated reports. To ease the process of data collecting, the script collects the data from all designs and stores it in separate files. The data is collected using *grep* commands, and the data is stripped of unnecessary text, using bash's substring replacement function, before output to files. The collected data is stored under the *reports*-directory, with the filenames:

From synthesis:
`register_count.rpt`

From Layout:
`combinational_area.rpt`
`noncombinational_area.rpt`
`design_area.rpt`

From power analysis:
`net_switching_power.rpt`
`cell_internal_power.rpt`
`cell_leakage_power.rpt`
`total_power.rpt`

Combined:
`all_results.rpt`

Each file contains the information specified by the filename. The corresponding design number is not included in the files, but the first line of each file contain results from design 0, the second line contain results from design 1, and so on. In the reports from power analysis, five values are stored at each line of the file, separated by commas. This is because the power analysis run five different power scenarios, each generating one result. The other files only contain a single value at each line. To simplify the process of importing the data into spreadsheets or other visualization-tools all files are joined horizontally into a single file, *all_results.rpt*. This means that each line will contain all values for a single design. The values in this file will be separated by tabs. This tidy file can be imported in Excel or other tools for generating graphs or compare data. Comparison could be made automatic if desired, but this is not implemented at this stage.

5.3 Running the framework

Before running the framework, some files need to be changed. The functional specification needs to be added to the file *designname.c* and testcases for the simulation can be added to the file *test_designname_testcases.v* under the directory *sim/tb/*.

In addition, the desired constraints need to be selected or filled out in the file *constraints.xlsx* in the *hls*-directory. To run the framework, the file *FrameworkScript.sh* is executed from a shell. Depending on the design size, available licences, and number of randomized constraints, the run-time of the framework can be long. If the framework seems stuck on one of the tasks, check the log file, *FrameworkScript.log*, for errors.

Chapter 6
Framework results

The framework is tested using the reference design of a FIR-filter, described in section 2.7.1. The source code of the filter, written in C, is listed in appendix A.1.1. The implemented FIR-filter has a 32-bit input, 16 taps and 64-bit output. The filter-coefficients are for simplicity defined to be the integers 1-16. By running this design through the framework, multiple results will be generated. These results will be compared towards each other, but also towards the same FIR-filter implemented directly in Verilog. The source code of the Verilog implementation is listed in appendix A.1.3. This testing will serve as a proof of concept, verifying the feasibility of the concept.

6.1 First test-run

The first test-run was performed with 6 randomized constraints input to LegUp. Each of the constraints could take values 1 and 0, giving a total of $2^6 = 64$ combinations. The randomized constraints and the pattern of how the values are set is shown in table 6.1.

Constraint	SDC NO CHAINING	PIPELINE ALL	MB MINIMIZE HW	ENABLE PATTERN SHARING	DUAL PORT BINDING	CASE FSM
Value	0	0	0	0	0	0
	0	0	0	0	0	1
	0	0	0	0	1	0
	0	0	0	0	1	1
	⋮	⋮	⋮	⋮	⋮	⋮
	1	1	1	1	1	0
	1	1	1	1	1	1

Table 6.1: Constraints and values for first run

This framework-run only include HLS, simulation, and synthesis, as the layout and power analysis tools have not yet been incorporated into the framework flow. Synthesis is run using a 32 MHz clock and a 180 nm cell library. A simple testbench generated by LegUp, with testcases similar to the ones listed in listing 4.7, were used for this run.

The presented results are gathered from the synthesis reports. The generated Verilog-code for many of the constraint files synthesize into the exact same area and estimated power consumption. This indicates that some of the combinations are redundant. The results are shown in table 6.2. As there are only 8 different results, only $log_2(8) = 3$ constraint parameters affect the design, the other will be don't-care constraints. By converting the design number to binary, a pattern can be seen. Table 6.3 shows the binary conversion of the design numbers at the second row of table 6.2. Here it can be seen that the parameters *PIPELINE_ALL*, *ENABLE_PATTERN_SHARING* and *DUAL_PORT_BINDING* are don't care for this design, since these parameters are not constant.

The area results are given in cell units, dynamic and total power are given in milliWatts (mW), and leakage power is given in nanoWatts (nW).

		Power		
Design #	**Area**	**Dynamic**	**Leakage**	**Total**
Verilog	175517.771501	2.9522	13.2559	2.9522
9,11,13,15,25,27,29,31	542636.067533	1.1933	445.5482	1.1937
8,10,12,14,24,26,28,30	543715.713936	1.1909	442.7919	1.1913
41,43,45,47,57,59,61,63	570759.857112	1.3097	474.8710	1.3102
1,3,5,7,17,19,21,23	571069.521032	1.2792	467.3262	1.2797
0,2,4,6,16,18,20,22	574368.902419	1.2745	467.3031	1.2750
40,42,44,46,56,58,60,62	574468.505099	1.3100	475.3570	1.3105
33,35,37,39,49,51,53,55	598731.305489	1.3951	498.4164	1.3956
32,34,36,38,48,50,52,54	599552.442242	1.3949	500.0916	1.3954

Table 6.2: Results from 1. framework-run

The best result with regards to area is the ones with the parameters *SDC_NO_CHAINING* set to 0, *MB_MINIMIZE_HW* set to 1 and *CASE_FSM* set to 1. The best result with regards to total power consumption is the same as for area, but with *CASE_FSM* set to 0.

The results are visualized in figure 6.1. It is clear from the graph that varying results are achieved from the different constraints. In figure 6.2, the best area-result from the framework is compared towards the results from the same design written in

Decimal	Binary
9	001001
11	001011
13	001101
15	001111
25	011001
27	011011
29	011101
31	011111

Table 6.3: Decimal to binary conversion of design numbers

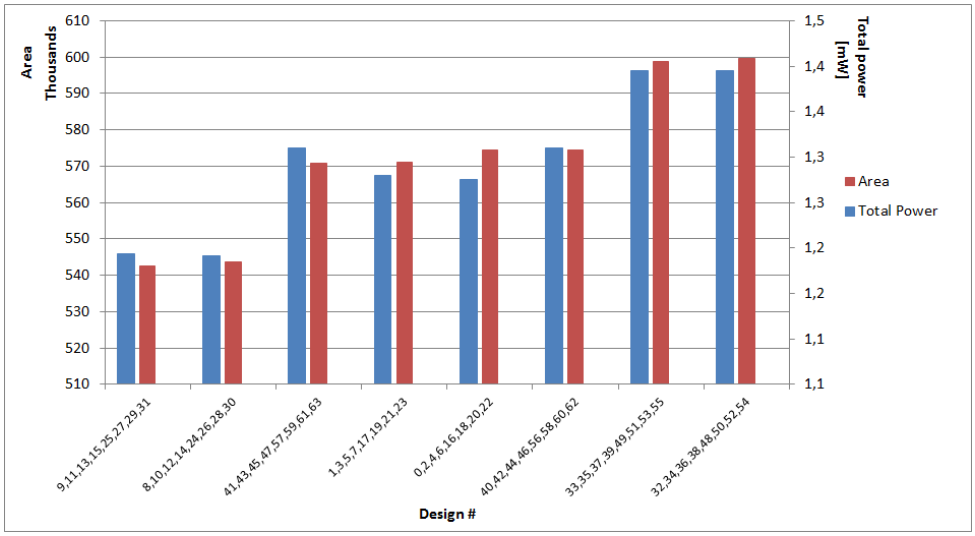


Figure 6.1: Results from 1. framework-run

Verilog directly. The design written directly in Verilog is better in terms of area, but it does not make sense that the estimated power consumption of the design written in Verilog is much higher than the HLS-generated design.

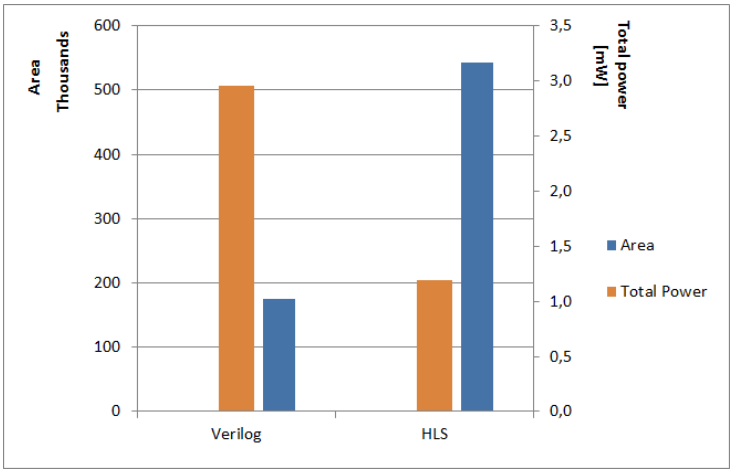


Figure 6.2: Comparison of Verilog-design towards best HLS-design from 1. framework-run

6.1.1 Handling unexpected results

It seems strange that the area consumption of the design written in Verilog is just 32.3% of the best result from LegUp, while the power consumption of the design written in Verilog is 247.4% of the best result from LegUp. Typically a larger design will consume more power, as each of the components has leakage and static operation consumption. Notice that these results were obtained using a static power analysis tool. The amount of switching in gates and registers have not been taken into account. However, this unexpected result needed to be investigated further. The following steps were taken to ensure the quality of the results to be acceptable:

- Check generated reports for misinterpreted data
- Look at schematic view of synthesized design to find errors
- Run HLS and synthesis once more to see if results deviates

None of the two first steps showed any errors. The designs are however too large to make any sense of the schematics, and due to to the limitation in setting signal sizes in the C-code, the HLS-generated designs scale down very poorly. The third step was performed on the same design, but with the clock relaxed from 32 MHz to 16

MHz. Changing the clock should not affect the design that much, but the synthesis will try to optimize the circuit to use the least amount of area, but still meet timing requirements. This time, only the constraints that had an impact on the design were included, generating 8 different designs. The result of the second run is shown in table 6.4 and in figure 6.3.

		Power		
Design #	**Area**	**Dynamic**	**Leakage**	**Total**
Verilog	175531.077145	1.0961	291.4545	1.0964
3	736221.630958	3.0417	600.7903	3.0423
2	738362.360388	3.0424	603.0359	3.0430
0	766630.553023	3.1736	627.3008	3.1742
1	783120.518084	3.2230	641.3857	3.2236
7	796905.744222	3.0702	634.3779	3.0708
6	798087.595262	3.0693	635.5286	3.0699
4	830305.921961	3.2036	660.1381	3.2043
5	853504.409992	3.2898	681.0897	3.2905

Table 6.4: Results from 2. framework-run

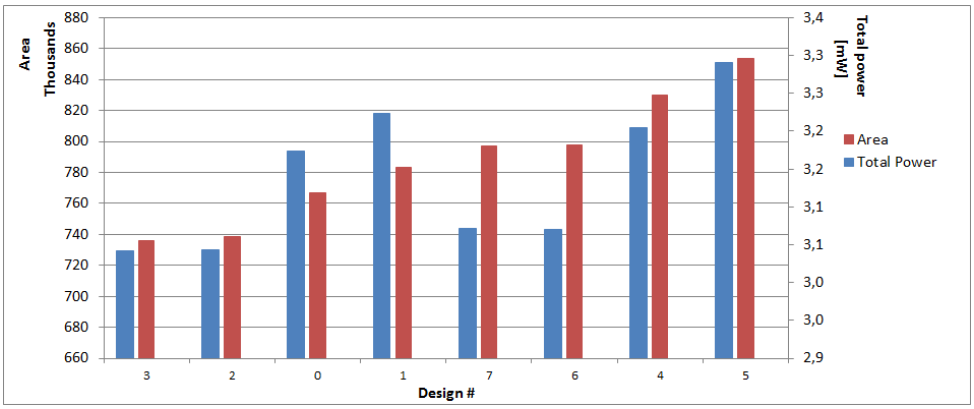


Figure 6.3: Results from 2. framework-run

These results are a better match with our expectations. Both area and power has increased a bit for all HLS-generated designs. The reason for this is not known for sure, but it is possible that synthesis use larger library-cells to meet timing requirements. All timing requirements were not met in the first run, as described in section 6.4, which could lead to synthesis giving up and reporting a smaller area. It could also have been a bug in the design or a wrong setting in the first run that

generated odd results. When comparing the best HLS-result towards the design written in Verilog, as shown in figure 6.4, we see that the Verilog design has both lower area and power consumption, which is what would be expected.

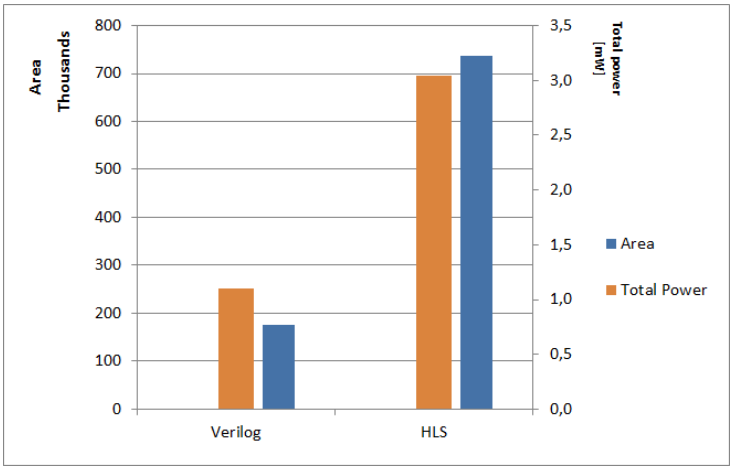


Figure 6.4: Comparison of Verilog-design towards best HLS-design from 2. framework-run

6.2 Full tool-flow framework run

The full tool-flow was incorporated into the framework, to see if the more accurate power analysis tool shows a different result. With the full tool-flow, the area reports are gathered from layout instead of synthesis and the power estimates are gathered from the power analysis. The full tool-flow is run using the same design as in section 6.1, with only the three constraints that affected the output, again a total of 8 designs. Since the full tool-flow use switching data from simulation to perform power analysis, the testbench used for simulation had to be changed to get more switching values. The new testbench applies 1000 random inputs to the input *inData*. It waits for the flag *iterationFinished* to be set before applying a new input. The full source code of this testbench is listed in appendix A.1.4. The testbench for the design written in Verilog is almost exacly the same, just adapted to the correct signal names and instead of waiting for a flag, *inData* is assigned a new value after 17 clock-cycles. The area results are shown in table 6.5 and the power estimates are shown in table 6.6.

In figure 6.5 the final area and power estimation results are visualized together. It is clear from the graphs that the concept works, as we get different results from the designs. These results are considered more accurate, as they use the reports from the chip layout for area, and power estimation using switching activity from

	Area		
Design #	**Combinational**	**Non-combinational**	**Total**
Verilog	72818.22296	44647.6792	117465.9022
3	130953.7155	201114.5108	332068.2263
2	131389.4738	201114.5108	332503.9847
0	136182.8164	210973.9603	347156.7766
1	139276.3681	213895.2786	353171.6467
7	136262.6501	220468.2449	356730.8950
6	137237.2853	220468.2449	357705.5302
4	141315.4518	230327.6944	371643.1462
5	146770.7475	236170.3311	382941.0786

Table 6.5: Area results from full tool-flow framework-run

	Power			
Design #	**Net switching**	**Internal**	**Leakage**	**Total**
Verilog	0.365	1.760	100.340	2.125
2	1.214	5.157	281.820	6.371
3	1.236	5.270	273.940	6.507
6	1.184	5.484	293.960	6.669
0	1.273	5.518	292.400	6.792
1	1.279	5.517	307.760	6.797
7	1.255	5.567	267.960	6.822
4	1.238	5.696	308.180	6.934
5	1.269	5.771	327.280	7.041

Table 6.6: Power estimation results from full tool-flow framework-run

simulation. Notice that compared to the results from synthesis, the area is actually cut by half, but the estimated power consumption has nearly doubled. This is because the layout tool can run optimizations on the design, decreasing the area, while the power estimates takes the dynamic switching activity into account, increasing the total power consumption. In figure 6.6, the area and power estimates from the full tool-flow framework-run is compared against the same design written directly in Verilog. Here we see the same trend as shown in the synthesis results above, but the relationship between area and power consumption shows more resemblance.

Figure 6.7 and 6.8 shows the distribution of area and power consumption within the designs. In the power graphs, leakage power is not shown as this is negligible

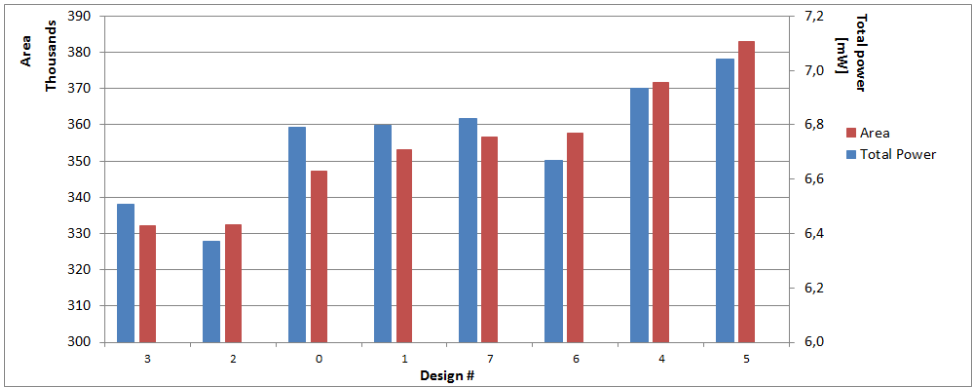


Figure 6.5: Results from framework with full tool-flow

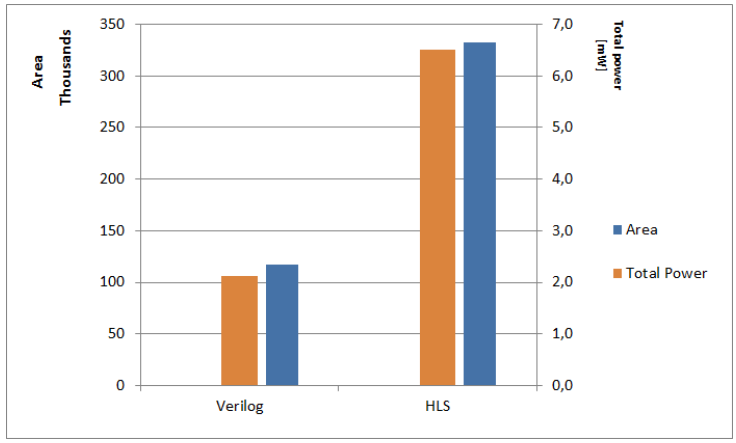


Figure 6.6: Comparison of Verilog-design towards best HLS-design from full tool-flow framework-run

compared to the other factors. The trend is the same in all the HLS generated designs, the larger portion of the area is consumed by non-combinational area. In the Verilog designs, the area distribution is inverted. This difference is what is expected from a FSM vs not-FSM design. In the power distribution graph we see that most of the power is consumed internally in the cells both in HLS generated designs and the Verilog design.

Two other interesting parameters to compare, is the number of generated registers, and the critical path and corresponding maximum frequency. For the implemented design, a minimum of 480 1-bit registers (`INPUTSIZE*(TAPS-1)`) is needed for the shift register, the *sr*-array. The number of registers, gathered from the synthesis reports, is shown in table 6.7.

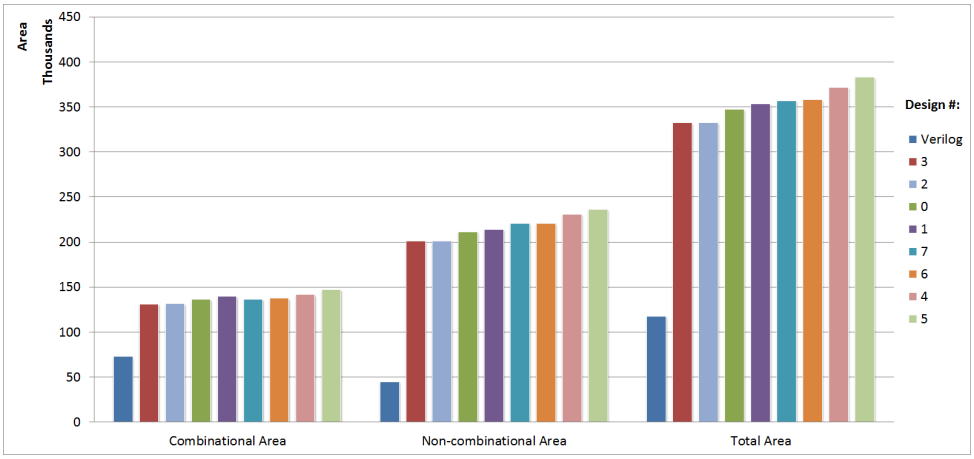


Figure 6.7: Area distribution of results from framework with full tool-flow

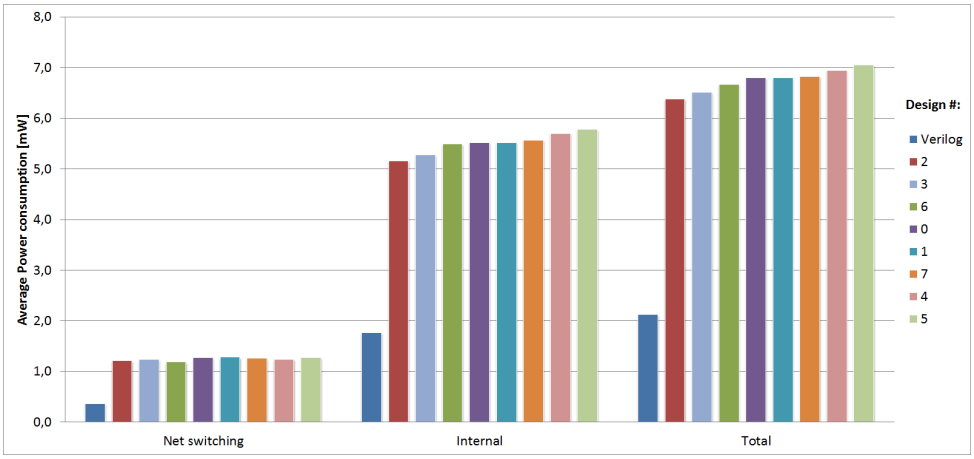


Figure 6.8: Power distribution of results from framework with full tool-flow

Design #	Registers
Verilog	480
0	2311
1	2343
2	2203
3	2203
4	2523
5	2587
6	2415
7	2415

Table 6.7: Number of used registers from full framework run

The maximum speed of the designs can be calculated from the critical path length, reported in both synthesis and layout. The critical path is the longest path through the circuit, calculated as the summation of the delay of each logical cell. The maximum frequency of the design is given as:

$$F_{MAX} = \frac{1}{t_{criticalPath}} \tag{6.1}$$

The critical path results and corresponding F_{MAX} is shown in table 6.8. The results for the HLS-generated designs are quite similar, but notice the large gap down to the design written directly in Verilog. Here the overhead of the FSM generated by LegUp is revealed. The synthesis and layout are setup to perform optimization of the area, while making sure the timing requirements of the target clock-period is met. This optimization goal can explain why the results are so similar for all designs.

6.3 Bugs in the generated design

To avoid the generation of a global memory controller, the flag *NO_INLINE* had to be set to 0 in the Makefile. This introduces two bugs in the generated Verilog; firstly, the signal generated from the parameter *done* is not sampled after the first time, making the program run forever, secondly the statement `products[0] = inData * 1` is only evaluated on the first iteration of the loop, meaning all outputs from the FIR-filter (except the first one) will have a deviation from the correct result, corresponding to `correctResult - inData + firstInData`. From the generated LLVM IR-code it looks like the first bug occurs because the comparison of the input *done* being performed under another label than where it is used as exit-condition for the while-loop. Under the same label, *inData* is sign-extended, as it is a 32-bit variable being assigned to a 64-bit variable. Both these results are stored to temporary registers

	Critical Path Length [nS]		F_{MAX} [MHz]	
Design #	**Synthesis**	**Layout**	**Synthesis**	**Layout**
Verilog	2.33	3.07	429.2	325.7
0	47.14	50.80	21.21	19.69
1	47.28	52.33	21.15	19.11
2	47.14	51.11	21.21	19.57
3	47.22	51.77	21.18	19.32
4	47.75	54.53	20.94	18.34
5	49.23	54.66	20.31	18.30
6	50.79	56.16	19.69	17.81
7	50.11	53.70	19.96	18.62

Table 6.8: Critical path length and maximum frequency results from full framework run

for use later, as shown below. From the simulation, it can be seen that the states generated by these operations are not visited again when the loop has been entered. It looks to be the implemented method for supporting streaming inputs and outputs that is the source of these bugs. When calling a function in C, it is not expected that the input-parameters shall change during run-time. It is therefore reasonable that the compiler schedule these operations before entering the while-loop, to prevent doing the same work multiple times.

```
1 %6 = icmp eq i32 %done, 0
2 %7 = sext i32 %inData to i64
```

These bugs are not critical to this proof of concept, as both bugs will be identical to every design. As these results are only supposed to show that the concept works, it is not critical that the generated results are accurate, as long as the fidelity of the results is high.

6.4 Path and hold violations

Some of the designs report violating path-length and hold-times in synthesis during 1. framework-run. For a real circuit this would be a problem. To get rid of these violations, the target clock speed during synthesis could be decreased to something below the maximum frequency of the design. However, the clock speed is most of the time the only thing you cannot change in your design, and the synthesis tool will try to create the circuit that meets timing. If timing is not met, the circuit description should be changed. This means that from our methodology, we will generate many circuits and only the ones that meet timing will be presented as an accepted solution.

Ideally, the framework would implement a feedback loop, but for the sake of this proof of concept, a long list of solutions are produced and only the best ones are selected. No violations were reported during 2. or 3. framework-run.

6.5 LegUp specific code optimization

When going trough the register-count from synthesis, it was noticed that the HLS-generated designs implemented both the array *sr* and *products* as RAM modules using registers. In the design written directly in Verilog, the *sr*-array is the only consumer of registers. When looking at the following snippet from the FIR-filter source code, listed in appendix A.1.1:

```
for (int k = 1; k < TAPS ; k++){
    products[k] = sr[k-1] * (k+1);
}
sum = sum + products[i];
```

it can easily be seen that this is functionally equivalent to:

```
sum = sum + (sr[i-1] * (i+1));
```

It can be argued that the second listing is better C-code, but it would be natural to assume modern compilers could handle such optimizations automatically. This optimization should be done in the compiler or in LegUp, but it was not done either places.

If we also substitute the code:

```
products[0] = inData * 1;
sum = products[0];
```

with:

```
sum = inData * 1;
```

the whole *products*-array can be removed. When running this optimized code through the framework, the result is much better than without these optimizations. Table 6.9 shows the results from the framework run of design 2, the best design with regards to power consumption from the full tool-flow framework-run.

The results from the optimized code gives the overhead shown in table 6.10. These overhead percentages corresponds more with the typical overhead of 30-40% in HLS-tools.

Synthesis	
Total Area	358587.628093
Net Switching Power	0.1340 mW
Internal Power	1.3203 mW
Leakage Power	283.1053 nW
Total Power	1.4546 mW
Register count	899

Layout	
Combinational Area	72482.256207
Non-combinational Area	82070.787659
Total Area	154553.043866

Power analysis	
Avg. Net Switching Power	0.604 mW
Avg. Internal Power	2.413 mW
Avg. Leakage Power	105.500 nW
Avg. Total Power	3.018 mW

Table 6.9: Results of best design from framework run with optimized C-code.

Overhead	
Layout area	37087.141666 (31.57%)
Average power	0.893mW (42.02%)
Register count	419 (87.29%)

Table 6.10: Overhead from results of optimized C-code.

Chapter 7
Discussion

The framework has been created to be easily configurable if it is desirable to add other functionality. For this proof of concept, only constraints set on the HLS process in LegUp has been used. It could also be useful to add other parameters to the framework, to control other parts of the tool-flow. Examples could be to run synthesis using different clock-speeds, or to specify optimization goals, like minimum area or maximum speed, for synthesis and layout. This would generate an even wider pool of designs to choose from, increasing the chance of getting the very best design. The downside of including more parameters into the framework is the exponential increase in the number of designs and accompanying tool-flow run-time. If all possibilities of 50 different constraints, each having two possible values, should be explored, a total of $2^{50} = 1\ 125\ 899\ 906\ 842\ 624$ (over 1 quadrillion) designs would have to be run through the tool-flow. If the tool-flow used 1 minute to process each design, the run-time for the framework would end up at 2.142 billion years. In practice, the designer is therefore required to select a few parameters that is assumed to have a large impact on the designs architecture, to get the best possible results from the framework.

From the results given in section 6.2, the best-case and worst-case design can be compared. A potential area saving of 50872.8523 and power saving of 0.670 mW can be achieved, by selecting the best architecture over the worst. This corresponds to a saving in area of 13.28% and a saving in power consumption of 9.52%. Compared to the design written directly in Verilog, the best-case area is 282.69% and power consumption is 299.81% of the results achieved in the Verilog design. An overhead of 182.69% and 199.81% are not great results, but the idea here is not to get a comparable result, rather to show that the concept works and can be used for a framework for architectural exploration. This goal has been achieved, as we get varied outputs depending on the given HLS constraints. As seen in section 6.5, the overhead generated by LegUp can be decreased greatly by optimizing the functional specification code. This gives an overhead of 31.57% in terms of area and 42.02% in terms of average power consumption. These numbers are much more in sync with

the expected overhead of HLS-tools of 30-40%. However, it would not always be this easy to see the potential optimizations that can be performed on the code. The same code was used in the design written directly in Verilog, but here the excessive registers were marked as redundant by synthesis and thus optimized away. The same optimization could not be performed on the HLS-generated designs, as synthesis could not draw the same conclusion through the FSM created by LegUp.

When looking at the speed of the generated designs compared to the design written in Verilog, there is a huge gap in the theoretical maximum frequency. This is a major drawback for the use of LegUp to produce hardware. There is also a second factor to the actual speed of the design. While the design written in Verilog primarily uses combinational logic to generate the output, the HLS-generated design use a FSM. In the combinational circuit, the same operation is performed each clock-cycle, leading to the finished calculation of the FIR-filter being available at the output-port 16 clock cycles after the input was set. In the HLS-generated design, a total time of $47840ns$ is required after the input is applied before the output is ready. This corresponds to 2392 clock-cycles with the clock-period of $20ns$ used under simulation. The actual time needed for calculation is then:

$$Verilog : 3.07\frac{ns}{cycle} * 16\ cycles = 49.12ns$$

$$HLS : 50.80\frac{ns}{cycle} * 2392\ cycles = 121513.6ns$$

This means that the design written in Verilog is 2473.81 times as fast as the HLS-generated design when it comes to producing a valid output, if both are run at their maximum frequency. If both designs were run at the 16 MHz clock-frequency target of synthesis, the Verilog-design would still be 149.5 times as fast as the HLS-generated design. Using the optimized code described in section 6.5, a total of 1235 clock-cycles is required to produce the output. The Verilog-design would then still be 1277.24 times as fast as the HLS-generated design. If both were run at the synthesis clock-frequency, the Verilog-design would be 77.19 times as fast as the HLS-generated design.

Even though the intended proof of concept has been shown, the alterations done to LegUp to get Verilog-output suitable for ASIC implementation has put constraints on how the functional specification can be written and limited the use of many features of the higher level of abstraction in C. Originally, the input was ANSI-C, supporting functions, arrays, structs, global variables, floating point, and pointers [18]. When using this adapted version of LegUp for generating ASIC-compliant Verilog, only functions and partially arrays seem to work correctly. Especially the pointer-, and related array-support, should be working correctly for the tool to be useful. Most of

these limitations can probably be overcome by altering the libraries further, but this will be time-consuming. One possible solution to bring some of these features back would be to re-introduce the *top*-module that instantiates the *main*-module and the *memory_controller*-module and connect all inputs and outputs directly to the ports of the *main*-module in *top*. This would work with the implementation of outputs from parameters and the streaming port feature. The downside of this approach is that the memory controller will bring back some extra overhead to the design. An alternative solution would be to further alter the libraries of LegUp, to make sure all RAMs are implemented as local RAMs.

There are still some bugs originating from the implemented alterations to LegUp present in the designs. These bugs were not focused on during the work with this thesis, but is something that must be resolved if the framework should be able to generate functional results. It is also an element of concern that the way you write your functional specification can affect the generated designs greatly. The compiler or LegUp should be able to optimize away any parts of excess code.

Chapter 8
Conclusion

This thesis has presented a solution for adapting the HLS-tool LegUp, to make the tool produce Verilog-output suitable for synthesis towards ASIC architectures. In addition to this, a framework for architectural exploration of digital hardware has been developed, using LegUp to increase the abstraction-level of a functional description. The framework is capable of generating a wide variety of architectural implementations of the given functional specification, using randomized constraints in the HLS-flow to get varying output. Each of the designs generated by HLS will automatically be simulated and synthesized, before layout and power analysis is performed on the design. The framework will generate reports containing relevant parameters, like area, power estimates, performance, and register count, for each design. The reports allow for the designer to easily compare the designs and select the one best suited based on the specification.

To ensure the functionality of the framework, a proof of concept has been conducted, using a FIR-filter as the reference design. The results from the framework shows that three of the six randomized constraint used, had an impact on the generated design. This gave a total of 8 architectural variations. By comparing the best and worst of the generated architectural variations, a decrease in area of 13.28% and a decrease in power consumption of 9.52% could be achieved. These results indicate that the proof of concept works.

When running the framework, some problems and bugs in the generated design were observed. Some of these were related to how the adaption of LegUp is performed, while some could be bugs in LegUp or the tool-flow. It is also an element of concern that the way you write your functional specification can affect the generated designs greatly. The compiler or LegUp should be able to optimize away any parts of the code that is obsolete. If this concept is to be used at a professional level, it is vital that all parts of the framework generate error-free results.

The overhead in area and power estimates are still high, with a best observed result

of 31.57% in terms of area and 42.02% in terms of average power consumption. The overhead is especially high in the non-optimized functional specification used in most of chapter 6, where the generated overhead in area and power consumption is close to 200%. This overhead needs to be reduced, or at least ensured to stay at a constant percentage, if the HLS-tool is to be used in the framework.

The final conclusion is that the concept have been shown and is generating varying results in the proof of concept. As only one design was used during the proof of concept, it cannot be concluded that the concept will work for every design until further testing has been conducted. More designs should be run through the framework to ensure the consistency of the results. It is also shown that LegUp, in the adapted version described in this thesis, can be used in a framework for architectural exploration. However, much more effort should be put into verification and testing of all parts of the tool-flow, to ensure all bugs and potential errors are eliminated before using it for any commercial purpose.

8.1 Future work

The following subsections will describe some areas of interest that is suggested to looked into in more detail if this thesis should be continued into a project or thesis at a later point.

8.1.1 Abstraction level

During the process of adapting LegUp, to make the generated Verilog more suitable for ASIC implementation, much of the higher level of abstraction originally supported by LegUp, have been lost. The limitation on how the code can be written, reduces the usefulness of the framework. It would increase the value of the framework greatly if a better solution to the resolved issues can be found, or another method can be used to bring back the desired functionality from ANSI-C.

8.1.2 Resolving bugs

In section 6.3, two bugs that were noticed during simulation have been described. For the framework to be useful, the generated designs should be bug-free, given that no bugs are present in the functional description. It should be put some effort into figuring out what is creating these bugs, and find a solution to avoid that these, or other, bugs are generated in future designs.

8.1.3 Eliminating RAM states

In section 3.6.3, a method of removing local RAMs generated by using the keyword *volatile* for input-parameters was described. This method of removing the RAMs

does not seem to remove the states designated to the allocation and storing to the generated RAMs. An example of this can be seen by looking at the generated state machine in figure 4.1. Here the states *1* and *2* does not perform any operations, as they were dedicated to allocation of, and storing to, the generated RAM. A method of removing these states should be added to the altered version of LegUp. Having states in the FSM that do not perform any operation, only leads to extra clock-cycles being used to produce the desired output, decreasing the overall speed of the circuit.

8.1.4 Advances in LegUp since last release

For this thesis, the current release version (4.0) of LegUp, released in August 2015, was used. LegUp's normal release cycle is roughly once a year [18]. From the GIT repository of LegUp [5], it can be seen that many new features have been, and still are being, implemented for the next release version of LegUp. One of the more interesting features from the views of this thesis is the implementation of streaming inputs and outputs. Even though this has been implemented in this thesis as well, a native implementation from the developers can be more thoroughly tested and have more functionality than the one implemented here. However, it is still uncertain if a good way of producing multiple output-signals have been implemented. In another commit it is mentioned that it will be possible to mark a RAM as external, making it possible to pass pointers as arguments into the top-level function. These features are exciting news from the perspective of this thesis, as it can look like the developers implements more things useful for ASIC hardware development. The upcoming release of LegUp, should be explored to see if any of the new features is useful for the concept described in this thesis.

8.1.5 Automatic code-optimization

The overhead in both area and power consumption could be reduced greatly by manually optimizing the code input to the framework, as described in section 6.5. This is a huge drawback of the framework, as it forces the designer to keep focus on writing the code in the most correct way, instead of focusing on writing the correct functional specification. The task of optimizing the input should be left to the compiler or HLS-tool. It would be strongly beneficial if a solution could be implemented to ensure that the code is optimized correctly.

8.1.6 Incorporating Nordic Semiconductors DDVC

The last objective described in section 1.3 have not been considered in this thesis, as the work with providing a functional framework and creating the proof of concept took all of the available time. It would still be interesting to see if the incorporation of Nordic Semiconductors DDVC into the Verilog-generating libraries of LegUp will reduce the overhead of the tool.

References

[1] Christiaan Baaij. Clash: From haskell to hardware. Master's thesis, University of Twente, 2009.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998. ISSN 0362-1340.

[4] Andrew Canis. High-Level Synthesis with LegUp. http://legup.eecg.utoronto.ca/, 2015. [Online; Accessed: 2016-06-09].

[5] Andrew Canis. LegUp GIT repository. http://legup.eecg.utoronto.ca/git?p=legup.git, 2016. [Online; Accessed: 2016-05-29].

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[7] clang. clang: a C language family frontend for LLVM. http://clang.llvm.org/, 2007. [Online; Accessed: 2016-06-05].

[8] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4): 8–17, 2009.

[9] Doxygen. LLVM API documentation: legup namespace reference. http://legup.eecg.utoronto.ca/doxygen/namespacelegup.html, 2011. [Online; Accessed: 2016-05-30].

[10] Doxygen. legup::rtlop class reference. http://legup.eecg.utoronto.ca/doxygen/classlegup_1_1RTLOp.html, 2011. [Online; Accessed: 2016-05-16].

[11] Mentor Graphics. Questa® advanced simulator. https://www.mentor.com/products/fv/questa/, 2015. [Online; Accessed: 2016-06-04].

[12] Haskell. Haskell: An advanced purely-functional programming language. https://www.haskell.org/, 2015. [Online; Accessed: 2016-06-05].

[13] Jørgen Frydenlund Holmefjord. High-level synthesis for hardware architectural exploration. In *Specialization project report*. Norwegian University of Science and Technology, December 2015.

[14] Chris Lattner. LLVM language reference manual. http://llvm.org/releases/2.7/docs/LangRef.html, 2010. [Online; Accessed: 2016-05-28].

[15] Chris Lattner. The architecture of open source applications: LLVM. http://www.aosabook.org/en/llvm.html, 2012. [Online; Accessed: 2016-06-05].

[16] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] LegUp. Constraints manual. http://legup.eecg.utoronto.ca/docs/4.0/constraints-manual.html, 2015. [Online; Accessed: 2016-06-05].

[18] LegUp. Frequently asked questions - LegUp 4.0 documentation. http://legup.eecg.utoronto.ca/docs/4.0/faq.html, 2015. [Online; Accessed: 2016-06-09].

[19] LegUp. Programmer's manual. http://legup.eecg.utoronto.ca/docs/4.0/programmermanual.html, 2015. [Online; Accessed: 2016-06-05].

[20] LegUp. LegUp on Xilinx FPGAs. http://legup.eecg.utoronto.ca/docs/4.0/xilinx.html, 2015. [Online; Accessed: 2016-06-04].

[21] Mac Developer Library. LLVM compiler overview. https://developer.apple.com/library/mac/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/, 2012. [Online; Accessed: 2016-06-05].

[22] LLVM. LLVM's analysis and transform passes. http://llvm.org/docs/Passes.html, 2016. [Online; Accessed: 2016-05-31].

[23] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[24] Simon Moore and Gregory Chadwick. The tiger "MIPS" processor. http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html, 2010. [Online; Accessed: 2016-06-05].

[25] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[26] Preeti Ranjan Panda, B. V. N. Silpa, Aviral Shrivastava, and Krishnaiah Gummidipudi. *Power-efficient System Design*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 9781441963871.

[27] John G. Proakis and Dimitris K. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications, 4th Edition*. Pearson Prentice Hall, 2007. ISBN 978-0-13-228731-9.

[28] Paul T. Robinson. Developer toolchain for PS4. http://llvm.org/devmtg/2013-11/slides/Robinson-PS4Toolchain.pdf, 2012. [Online slides; Accessed: 2016-06-05].

[29] Per-Carsten Skoglund and Christoffer Amlo. Digital design and verification conventions (DDVC). [NORDIC INTERNAL; Confidential; Accessed: 2016-06-05], 2011.

[30] Wilson Snyder. Verilog-perl overview. http://www.veripool.org/projects/verilog-perl/, 2016. [Online; Accessed: 2016-05-31].

[31] Reid Spencer. Bug 1284 - support bitwidth attribute in llvm-gcc. https://llvm.org/bugs/show_bug.cgi?id=1284, 2007. [Online; Accessed: 2016-05-27].

[32] Synopsys. Design compiler 2010. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx, 2016. [Online; Accessed: 2016-06-04].

[33] Synopsys. IC compiler. http://www.synopsys.com/Tools/Implementation/PhysicalImplementation/Pages/ICCompiler.aspx, 2016. [Online; Accessed: 2016-06-04].

[34] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog HDL. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[35] Joar Nikolai Talstad. Channel filter cross-layer optimization. Master's thesis, Norwegian University of Science and Technology, 6 2015.

Appendix A
Source code listings

A.1 FIR-filter reference design

A.1.1 C source code

```
#include <stdint.h>

#define INPUTSIZE 32
#define TAPS 16

int main(int done, int inData, volatile long long int __out_outData) {

  int sr[TAPS-1] = {0};
  long long int products[TAPS] = {0};
  long long int sum = 0;
  while(done == 0){
    products[0] = inData * 1;
    sum = products[0];
    for (int i = 1; i < TAPS ; i++){
      for(int j = TAPS-1; j > 0; j--){
        sr[j] = sr[j -1];
      }
      sr[0] = inData;
      for (int k = 1; k < TAPS ; k++){
        products[k] = sr[k-1] * (k+1);
      }
      sum = sum + products[i];
    }
    __out_outData = sum;
  }
  return 0;
}
```

A.1.2 Optimized C source code

```
#include <stdint.h>

#define INPUTSIZE 32
#define TAPS 16

int main(int done, int inData, volatile long long int __out_outData) {
  int sr[TAPS-1] = {0};
  long long int sum = 0;
  while(done == 0){
    sum = inData;
    for (int i = 1; i < TAPS ; i++){
      for(int j = TAPS-1; j > 0; j--){
        sr[j] = sr[j -1];
      }
      sr[0] = inData;
      sum = sum + sr[i-1] * (i+1);
    }
    __out_outData = sum;
  }
  return 0;
}
```

A.1.3 Verilog source code

```
module fir (
    clk,
    reset,
    dataIn,
    dataOut
  );
  parameter WIDTH = 32;
  parameter DEPTH = 16;


  input clk, reset;
  input signed [WIDTH-1:0] dataIn;
  output wire signed [2*WIDTH-1:0] dataOut;
  integer i, j, k;
  reg signed [WIDTH-1:0] sr [DEPTH-2:0];
  reg signed [2*WIDTH-1:0] products [DEPTH-1:0];
  reg signed [2*WIDTH-1:0] sum;

  always @( posedge clk or posedge reset ) begin
    if( reset == 1) begin
      sum = 0;
    end
    else begin
    for(i = DEPTH-2; i > 0; i=i-1) begin
        sr[i] <= sr[i-1];
      end
    sr[0] <= dataIn;
    end
  end

  always @(*) begin
  products[0] = dataIn * 1;
    for (j = 1; j < DEPTH ; j=j+1) begin
      products[j] = sr[j-1] * (j+1);
  end
  end

  always @(*) begin
    sum = products [0];
    for (k = 1; k < DEPTH ; k=k+1) begin
      sum = sum + products[k];
  end
  end

  assign dataOut = sum;

endmodule
```

A.1.4 Testbench for FIR-filter

```
`timescale 1 ns / 1 ns
module test_fir
(
);

reg  clk;
reg  reset;
reg  start;
wire [63:0] return_val;
wire  finish;
reg  memory_controller_waitrequest;
reg [31:0] arg_done;
reg [31:0] arg_inData;
wire [63:0] arg_outData;
wire  arg_outData_valid;
wire  iterationFinish;


fir u_fir (
  .clk (clk),
  .reset (reset),
  .start (start),
  .finish (finish),
  .memory_controller_waitrequest (memory_controller_waitrequest),
  .return_val (return_val),
  .arg_done (arg_done),
  .arg_inData (arg_inData),
  .arg_outData (arg_outData),
  .arg_outData_valid (arg_outData_valid),
  .iterationFinish (iterationFinish)
);

// Clock generation
initial
    clk = 0;
always @(clk)
    clk <= #10 ~clk;

initial begin
@(negedge clk);
reset <= 1;
@(negedge clk);
reset <= 0;
start <= 1;
@(negedge clk);
start <= 0;
end

always@(finish) begin
    if (finish == 1) begin
        $display("At t=%t simulation finished", $time);
        $display("Cycles: %d", ($time-50)/20);
        $finish;
    end
end

initial begin
memory_controller_waitrequest <= 1;
@(negedge clk);
@(negedge clk);
memory_controller_waitrequest <= 0;
```

```
62 end
63
64 // Custom testcases:
65   initial begin
66     arg_inData = 32'b0;
67     arg_done = 32'b0;
68   end
69
70   initial
71   begin : TEST_CASE
72     @(posedge reset)
73     repeat (1000) begin
74       @(negedge clk);
75       arg_inData = $random;
76       @(posedge iterationFinish);
77     end
78     $display("Finished applying inData\n");
79     arg_done = 32'b1;
80     #100ns
81     $finish;
82   end
83
84 endmodule
```

A.2 LLVM IR Parser program

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <vector>
#include <set>
using namespace std;

int main(int argc, char *argv[]) {

    // Check for correct amount of arguments
    if (argc < 3) {
        cout << "Missing output file argument\n";
        if (argc < 2) {
            cout << "Missing input file argument\n";
        }
        cout << "Arguments should be at the form: inputfile outputfile\n";
        return 1;
    }

    vector<string> sources;
    vector<string> targets;
    vector<int> labels;

    ifstream inFile(argv[1]);
    ofstream outFile(argv[2]);

    if (inFile.is_open()) {
        cout << "inFile opened successfully\n";
        string line;
        string searchStringStores = "  store";
        string searchStringMain = "define";
        string labelString = "; <label>:";
        string whitespace = " ";
        int inMain = false;
        int currLabel = 0;
        bool isTarget = false;
        // Read file line by line
        while (getline(inFile, line)) {
            // Only consider lines staring with "  store"
            if (line.compare(0, searchStringMain.length(), searchStringMain) ==
                    0 &&
                !inMain) {

                size_t found = 0;
                do {
                    found = line.find(whitespace, found + 1);
                } while (line.compare(found + 1, 1, "@") != 0);
                if (line.compare(found + 1, 6, "@main(") == 0) {
                    inMain = true;
                    currLabel = 0;
                }
            } else if (line.compare(0, labelString.length(), labelString) ==
                       0) {
                currLabel = atoi(line.substr(10, 3).c_str());
            } else if (line.compare(0, searchStringStores.length(),
                                    searchStringStores) == 0 &&
                       inMain) {

                // Remove commas from line
```

```
61                  line.erase(std::remove(line.begin(), line.end(), ','),
62                             line.end());
63
64                  // Remove leading and trailing whitespaces
65                  line.erase(
66                      line.begin(),
67                      std::find_if(line.begin(), line.end(),
68                                   bind1st(std::not_equal_to<char>(), ' ')));
69
70                  // Split line at whitespace
71
72                  size_t found = line.find(whitespace);
73                  while (found != string::npos) {
74                      size_t foundNext = line.find(whitespace, found + 1);
75
76                      // Only store words staring with a % sign
77                      if (line.compare(found + 1, 1, "%") == 0) {
78                          string substring =
79                              line.substr(found + 2, foundNext - found - 2);
80                          if (isTarget == true) {
81                              targets.push_back(substring);
82                              labels.push_back(currLabel);
83                              isTarget = false;
84                          } else {
85                              sources.push_back(substring);
86                              isTarget = true;
87                          }
88                      }
89                      found = foundNext;
90                  }
91                  if (sources.size() > targets.size()) {
92                      sources.pop_back();
93                      isTarget = false;
94                  }
95                  if (line.compare(0, 1, "}") == 0) {
96                      inMain = false;
97                  }
98              }
99          }
100         inFile.close();
101     }
102
103     else
104         cout << "Unable to open input file\n";
105
106     if (outFile.is_open()) {
107         cout << "outFile opened successfully\n";
108         set<string> done;
109
110         // Iterate through all found stores and check for assignment ↩
    connections
111         for (int i = 0; i < targets.size(); ++i) {
112             for (int j = 0; j < targets.size(); ++j) {
113                 if (targets[i] == targets[j] && i != j &&
114                     done.find(sources[i]) == done.end() &&
115                     sources[i].find("__out_") == 0) {
116                     done.insert(sources[j]);
117                     string sigName = sources[i];
118                     // Only print parameters defined as outputs
119                     if (sigName.find("__out_") == 0) {
120                         sigName = sigName.substr(6, std::string::npos);
121                         outFile << sigName << " " << sources[j] << " "
```

```
122                                 << labels[j] << " " << labels[i] << " "
123                                 << targets[i] << "\n";
124                     }
125                 }
126             }
127         }
128         outFile.close();
129     }
130
131     else
132         cout << "Unable to open output file\n";
133
134     return 0;
135 }
```

A.3 Generating valid signals

```
// Add each driving signal from source as a driver of the target
// signal.
// Also generate conditions for valid signals and drive these.
for (uint j = 1; j < sourceSig->getNumDrivers(); j += 2) {
    if (sourceSig->getDriver(j)->getName().compare(
            targetSig->getName()) != 0) {
        targetSig->addCondition(sourceSig->getCondition(j),
                                sourceSig->getDriver(j));
        if (j + 1 < sourceSig->getNumDrivers()) {
            targetSig->addCondition(sourceSig->getCondition(j + 1),
                                    sourceSig->getDriver(j + 1));
        }
        if (sourceSig->getCondition(j)->isOp()) {
            validSig->addCondition(sourceSig->getCondition(j), ONE);
            if (j + 1 < sourceSig->getNumDrivers()) {
                validSig->addCondition(
                    sourceSig->getCondition(j + 1), ONE);
            }
            if (sourceSig->getNumDrivers() - 1 < 2) {
                // Adds deassertion of validSig if only single
                // conditions are present.
                validSig->addCondition(
                    rtl->addOp(RTLOp::Not)
                        ->setOperands(sourceSig->getCondition(j)),
                    ZERO);
            } else if (sourceSig->getNumDrivers() - 1 < 3 ||
                       j == 1) {
                notValid->setOperands(
                    rtl->addOp(RTLOp::Not)
                        ->setOperands(sourceSig->getCondition(j)),
                    rtl->addOp(RTLOp::Not)->setOperands(
                        sourceSig->getCondition(j + 1)));
            } else if (sourceSig->getNumDrivers() - j > 1) {
                RTLSignal *notValid1 =
                    rtl->addOp(RTLOp::And)->setOperands(
                        rtl->addOp(RTLOp::Not)->setOperands(
                            sourceSig->getCondition(j)),
                        rtl->addOp(RTLOp::Not)->setOperands(
                            sourceSig->getCondition(j + 1)));
                RTLSignal *notValid2 =
                    rtl->addOp(RTLOp::And)
                        ->setOperands(notValid->getOperand(0),
                                      notValid->getOperand(1));
                notValid->setOperands(notValid1, notValid2);
            } else {
                RTLSignal *notValid1 =
                    rtl->addOp(RTLOp::And)
                        ->setOperands(notValid->getOperand(0),
                                      notValid->getOperand(1));
                notValid->setOperands(
                    notValid1, rtl->addOp(RTLOp::Not)->setOperands(
                                   sourceSig->getCondition(j)));
            }
        }
    }
}
// Adds deassertion of validSig if multiple conditions are present.
if (notValid->getNumOperands() > 1) {
    validSig->addCondition(notValid, ZERO);
}
```

A.4 Adding iterationFinish flag

```
RTLSignal *interationFinish = rtl->addOutReg("interationFinish");

connectSignalToDriverInState(interationFinish, ONE, (--fsm->end())->getPrevNode());
interationFinish->addCondition(rtl->addOp(RTLOp::Not)->setOperands(interationFinish->getCondition(0)), ZERO);
```

A.5 Testbench generator source code

```
RTLModule *t = m->addModule("main", "main_inst");
if (LEGUP_CONFIG->getParameterInt("ASIC_IMPLEMENTATION")) {
  RTLModule *rtl = alloc->getModuleForFunction(alloc->getModule()->getFunction("main"));
  if (rtl->getName().compare("main") == 0) {
    for (RTLModule::const_signal_iterator i = rtl->port_begin(), e = rtl->port_end(); i != e; ++i) {
      const RTLSignal *s = *i;
      RTLSignal *d;
      string type = s->getType();
      if (!type.empty()) {
        if (type.compare(0, 6, "output") == 0) {
          d = m->addWire(s->getName(), s->getWidth());
          t->addOut(s->getName(), s->getWidth())->connect(d);
        } else {
          d = m->addReg(s->getName(), s->getWidth());
          t->addIn(s->getName(), s->getWidth())->connect(d);
        }
      }
    }
  }
}
```

A.6 Script for creating new framework-project

```
#!/bin/bash

echo "Type the name of the new design, followed by [ENTER]:"
read DESIGNNAME

LEVEL=$(pwd)
mkdir $DESIGNNAME
mkdir $DESIGNNAME/ip
mkdir $DESIGNNAME/ip/$DESIGNNAME
cp -r _source/methodology $DESIGNNAME
cp -r _source/ip/libs $DESIGNNAME/ip
cp -r _source/ip/designname/* $DESIGNNAME/ip/$DESIGNNAME
cd $DESIGNNAME/ip/$DESIGNNAME
mv designname.c $DESIGNNAME.c
mv rtl/designname.fl rtl/$DESIGNNAME.fl
mv rtl/designname_sim.fl rtl/$DESIGNNAME\_sim.fl
mv sim/tb/test_designname.fl sim/tb/test_$DESIGNNAME.fl
mv sim/tb/test_designname_testcases.v sim/tb/test_$DESIGNNAME\_testcases.v
mv sim/tb/test_designname.v sim/tb/test_$DESIGNNAME.v
mv sim/run/designname.args sim/run/$DESIGNNAME.args
mv sim/run/designname.comp sim/run/$DESIGNNAME.comp
mv sim/run/designname.sim sim/run/$DESIGNNAME.sim
mv syn/dc_scripts/designname.constraints.tcl syn/dc_scripts/$DESIGNNAME.↩
    constraints.tcl

find FrameworkScript.sh -type f -exec sed -i "s/DESIGNNAME=designname/↩
    DESIGNNAME=$DESIGNNAME/g" {} \;
find FrameworkScript.sh -type f -exec sed -i "s?basedir?$LEVEL?g" {} \;
find Makefile -type f -exec sed -i "s?basedir?$LEVEL?g" {} \;
find Makefile -type f -exec sed -i "s/designname/$DESIGNNAME/g" {} \;

find rtl/$DESIGNNAME.fl -type f -exec sed -i "s/designname/$DESIGNNAME/g" {} \;
find rtl/$DESIGNNAME\_sim.fl -type f -exec sed -i "s/designname/$DESIGNNAME/g" ↩
    {} \;
find sim/tb/test_$DESIGNNAME.fl -type f -exec sed -i "s/designname/$DESIGNNAME/↩
    g" {} \;

find sim/run/$DESIGNNAME.args -type f -exec sed -i "s/designname/$DESIGNNAME/g"↩
     {} \;
find sim/run/$DESIGNNAME.comp -type f -exec sed -i "s/designname/$DESIGNNAME/g"↩
     {} \;
find sim/run/$DESIGNNAME.sim -type f -exec sed -i "s/designname/$DESIGNNAME/g" ↩
    {} \;
find sim/run/$DESIGNNAME.sim -type f -exec sed -i "s?basedir?$LEVEL?g" {} \;
find sim/run/RUN_ALL -type f -exec sed -i "s/designname/$DESIGNNAME/g" {} \;

find syn/common_setup.tcl -type f -exec sed -i "s/designname/$DESIGNNAME/g" {} ↩
    \;
find syn/dc_scripts/dc_compile.tcl -type f -exec sed -i "s?basedir?$LEVEL?g" {}↩
     \;
```

A.7 Script for running framework

```
#!/bin/bash
rm -f FrameworkScript.log
LOG_FILE=FrameworkScript.log
exec 3>&1 1>>${LOG_FILE} 2>&1 # Print log to file, print specified echos to terminal

DESIGNNAME=designname
REMOTEIP=192.168.12.33 # IP of the computer running the LegUp VirtualBox guest
REMOTEPORT=3022 # Port that is forwarded to port 22 on VirtualBox guest
REMOTEDIR=/home/legup/legup-4.0/examples
LEGUPUSER=legup # Username of LegUp image
LEGUPPASS=letmein # Password of LegUp image
BASE_DIR=basedir
LOCALDIR=$BASE_DIR/$DESIGNNAME/ip/$DESIGNNAME #Location of source files on Linux server.

export DESIGN_NAME=$DESIGNNAME
export FILE_LIST=$DESIGNNAME
export BASE_DIR=$BASE_DIR
export VC_WORKSPACE=$BASE_DIR/$DESIGNNAME

module load icc # Load IC compiler module
module load primetime # Load PrimeTime module

SSHCOMMANDS2="mkdir $REMOTEDIR/$DESIGNNAME; cd $REMOTEDIR/$DESIGNNAME/; libreoffice --headless --convert-to csv constraints.xlsx --outdir .; exit" # ssh commands for converting excel file to csv

if [$1 = "-s"]; then
  echo Setup started
  ssh-keygen -f id_rsa -t rsa -N ''
  spawn ssh-copy-id "$LEGUPUSER@$REMOTEIP -p $REMOTEPORT"
  expect "password:"
  send "$LEGUPPASS\n"
  expect eof
  echo Setup finished
fi

mkdir -p $LOCALDIR/hls/
ssh $LEGUPUSER@$REMOTEIP -p $REMOTEPORT "mkdir -p $REMOTEDIR/$DESIGNNAME"
scp -P $REMOTEPORT $LOCALDIR/$DESIGNNAME.c $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME #Copy design file to LegUp image
scp -P $REMOTEPORT $LOCALDIR/sim/tb/test_$DESIGNNAME\_testcases.v $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME #Copy testcases file to LegUp

scp -P $REMOTEPORT $LOCALDIR/hls/constraints.xlsx $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/ #Copy design constraint definitions to LegUp image
ssh $LEGUPUSER@$REMOTEIP -p $REMOTEPORT $SSHCOMMANDS2 #Run commands and script for generating constraint and Makefiles
scp -P $REMOTEPORT $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/constraints.csv $LOCALDIR/hls/ #Copy CSV file from LegUp image
sed 's/\''//g' -i $LOCALDIR/hls/constraints.csv # Remove excess quotes
rm -r $LOCALDIR/hls/makefiles $LOCALDIR/hls/constraintfiles
mkdir $LOCALDIR/hls/makefiles $LOCALDIR/hls/constraintfiles
mkdir $LOCALDIR/reports
rm $LOCALDIR/reports/*.rpt
cd $LOCALDIR/hls/

$LOCALDIR/hls/constraintsGenerator.run $LOCALDIR/hls/constraints.csv .. $DESIGNNAME
```

```
NUMRUNS=$?
echo "Generated $NUMRUNS constraint and Makefiles" | tee /dev/fd/3
COUNTER=0
while [ $COUNTER -lt $NUMRUNS ]; do
  echo "Framework loop #$COUNTER" 1>&3
  rm $LOCALDIR/rtl/{*.tcl,*.v,*.mif}
  SSHCOMMANDS="export PATH=/home/legup/clang+llvm-3.5.0-x86_64-linux-gnu/bin:$PATH; cd $REMOTEDIR/$DESIGNNAME/; make clean; make; exit" # Commands to run on SSH session. Need to add clang to PATH as this is not present in SSH session.
  scp -P $REMOTEPORT $LOCALDIR/hls/constraintfiles/config$COUNTER.tcl $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/ #Copy design constraint file to LegUp image
  scp -P $REMOTEPORT $LOCALDIR/hls/makefiles/Makefile$COUNTER $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/Makefile #Copy design Makefile to LegUp image
  echo "Running HLS" 1>&3
  ssh $LEGUPUSER@$REMOTEIP -p $REMOTEPORT $SSHCOMMANDS #Run LegUp

  scp -P $REMOTEPORT $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/$DESIGNNAME.v $LOCALDIR/rtl/ #Copy Verilog file from LegUp image
  scp -P $REMOTEPORT $LEGUPUSER@$REMOTEIP:$REMOTEDIR/$DESIGNNAME/test_main.v $LOCALDIR/sim/tb/test_$DESIGNNAME.v #Copy Verilog testbench file from LegUp image

  MEM_CRTL_EXIST=$(grep -c "module memory_controller" $LOCALDIR/rtl/$DESIGNNAME.v)
  if [ $MEM_CRTL_EXIST -gt 0 ]; then
    echo "memory_controller module exist in design. Please check your design" 1>&3
  fi

  find $LOCALDIR/rtl/$DESIGNNAME.v -type f -exec sed -i "s/module main/module $DESIGNNAME/g" {} \; #Replace top modulename main with designname

  find $LOCALDIR/sim/tb/test_$DESIGNNAME.v -type f -exec sed -i "s/module main_tb/module test_$DESIGNNAME/g" {} \; #Replace tb declaration with correct designname
  find $LOCALDIR/sim/tb/test_$DESIGNNAME.v -type f -exec sed -i "s/main main_inst/$DESIGNNAME u_$DESIGNNAME/g" {} \; #Replace top module instantiation in tb with correct designname

  echo "Running simulation" 1>&3
  #Run simulation
  (cd $LOCALDIR/sim/run/ && (RUN_ALL --clean) && (vcd2saif -input $LOCALDIR/sim/run/$DESIGNNAME.vcd -output $LOCALDIR/sim/run/$DESIGNNAME.saif))

  echo "Running synthesis" 1>&3
  #Run synthesis
  (cd $LOCALDIR/syn/ && (make clean) && (make compile)) #Run synthesis clean removes old data

  echo "Running layout" 1>&3
  #Run layout
  (cd $LOCALDIR/lay/ && (make clean) && (make outputs_cts))

  echo "Running power analysis" 1>&3
  #Run power estimation
  (cd $LOCALDIR/pow/ && (make clean) && (make power_analysis))

  #Store synthesis results to common file

```

```
94  echo "Gathering layout results" 1>&3
95  var1=$(grep "Combinational Area:" $LOCALDIR/lay/reports/clock_opt_cts_icc.qor↩
      )
96  var1=${var1//  Combinational Area:/}
97  var1=${var1// /}
98  var1=${var1//./,}
99  echo $var1 >> $LOCALDIR/reports/noncombinational_area.rpt
100 var2=$(grep "Noncombinational Area:" $LOCALDIR/lay/reports/clock_opt_cts_icc.↩
      qor)
101 var2=${var2//  Noncombinational Area:/}
102 var2=${var2// /}
103 var2=${var2//./,}
104 echo $var2 >> $LOCALDIR/reports/combinational_area.rpt
105 var3=$(grep "Design Area:" $LOCALDIR/lay/reports/clock_opt_cts_icc.qor)
106 var3=${var3//  Design Area: /}
107 var3=${var3// /}
108 var3=${var3//./,}
109 echo $var3 >> $LOCALDIR/reports/design_area.rpt
110 var4=$(grep "Total number of registers" $LOCALDIR/syn/reports/$DESIGNNAME.↩
      mapped.clock_gating.rpt)
111 var4=${var4//          |    Total number of registers          |/}
112 var4=${var4// /}
113 var4=${var4//|/}
114 echo $var4 >> $LOCALDIR/reports/register_count.rpt
115
116 echo "Gathering power analysis results" 1>&3
117
118 COUNT=0
119 while [ $COUNT -lt 4 ]; do
120   swpow=$(grep 'Net Switching Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_ctrl$COUNT/power_summary.rpt)
121   swpow=${swpow//([^)]*)/}
122   swpow=${swpow//  Net Switching Power  = /}
123   echo -n "$swpow\t">>$LOCALDIR/reports/net_switching_power.rpt
124   intpow=$(grep 'Cell Internal Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_ctrl$COUNT/power_summary.rpt)
125   intpow=${intpow//([^)]*)/}
126   intpow=${intpow//  Cell Internal Power  = /}
127   echo -n "$intpow\t">>$LOCALDIR/reports/cell_internal_power.rpt
128   leakpow=$(grep 'Cell Leakage Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_ctrl$COUNT/power_summary.rpt)
129   leakpow=${leakpow//([^)]*)/}
130   leakpow=${leakpow//  Cell Leakage Power   = /}
131   echo -n "$leakpow\t">>$LOCALDIR/reports/cell_leakage_power.rpt
132   totpow=$(grep 'Total Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_ctrl$COUNT/power_summary.rpt)
133   totpow=${totpow//([^)]*)/}
134   totpow=${totpow//Total Power            = /}
135   echo -n "$totpow\t">>$LOCALDIR/reports/total_power.rpt
136   let COUNT=COUNT+1
137 done
138
139 swpow=$(grep 'Net Switching Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_inactive/power_summary.rpt)
140 swpow=${swpow//([^)]*)/}
141 swpow=${swpow//  Net Switching Power  = /}
142 echo $swpow>>$LOCALDIR/reports/net_switching_power.rpt
143 intpow=$(grep 'Cell Internal Power' $LOCALDIR/pow/reports/↩
      power_analysis_$DESIGNNAME\_inactive/power_summary.rpt)
144 intpow=${intpow//([^)]*)/}
145 intpow=${intpow//  Cell Internal Power  = /}
146 echo $intpow>>$LOCALDIR/reports/cell_internal_power.rpt
```

```
147   leakpow=$(grep 'Cell Leakage Power' $LOCALDIR/pow/reports/↩
        power_analysis_$DESIGNNAME\_inactive/power_summary.rpt)
148   leakpow=${leakpow//([^)]*)/}
149   leakpow=${leakpow//  Cell Leakage Power   = /}
150   echo $leakpow>>$LOCALDIR/reports/cell_leakage_power.rpt
151   totpow=$(grep 'Total Power' $LOCALDIR/pow/reports/power_analysis_$DESIGNNAME\↩
        _inactive/power_summary.rpt)
152   totpow=${totpow//([^)]*)/}
153   totpow=${totpow//Total Power            = /}
154   echo $totpow>>$LOCALDIR/reports/total_power.rpt
155
156   echo "Register count\tCombinational Area\tNon-combinational Area\tDesign Area↩
        \tSwitching Power\tInternal Power\tLeakage Power\tTotal Power" > ↩
        all_results.rpt
157   paste register_count.rpt combinational_area.rpt noncombinational_area.rpt ↩
        design_area.rpt net_switching_power.rpt cell_internal_power.rpt ↩
        cell_leakage_power.rpt total_power.rpt >> all_results.rpt
158
159   #Store results in dedicated folder
160   rm -f $LOCALDIR/sim/run/$DESIGNNAME.vcd #VCD file can get large. Remove ↩
        before storing framework run data.
161   mkdir -p $LOCALDIR/hls/rtl$COUNTER/
162   cp $LOCALDIR/hls/constraintfiles/config$COUNTER.tcl $LOCALDIR/rtl/ #Copy ↩
        design constraint file to current rtl folder
163   cp $LOCALDIR/hls/makefiles/Makefile$COUNTER $LOCALDIR/rtl/Makefile
164   cp -r $LOCALDIR/{rtl/,sim/,syn/,lay/,pow/,score/} $LOCALDIR/hls/rtl$COUNTER/
165
166   let COUNTER=COUNTER+1
167 done
168 echo HLS finished
169 exit $?
```

A.8 Constraint-generator program

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <math.h>
#include <sstream>
#include <map>
#include <vector>

using namespace std;

map<string, string> requiredConstraints;
vector<string> randomConstraints;
map<string, string> staticConstraints;
map<string, string> makefileConstraints;
map<string, string> nonParameterConstraints;

int main(int argc, char *argv[]) {

    // Check for correct amount of arguments
    if (argc < 4) {
        cout << "Missing design-name argument\n";
        if (argc < 3) {
            cout << "Missing Makefile LEVEL argument\n";
            if (argc < 1) {
                cout << "Missing constraints csv-fileName argument\n";
            }
        }
        cout << "Arguments should be at the form: csv-fileName LEVEL "
                "design-name\n";
        return 0;
    }

    // Read constraints from .csv file
    ifstream csvFile;
    csvFile.open(argv[1]);

    while (csvFile) {
        string s;
        if (!getline(csvFile, s))
            break;

        istringstream ss(s);
        vector<string> record;

        while (ss) {
            string s;
            if (!getline(ss, s, ','))
                break;
            record.push_back(s);
        }
        bool required = false;
        bool isParameter = false;
        bool isMakefile = false;
        string value = record.back();
        record.pop_back();
        if (value == "discard") {
            continue;
        }
        if (value == "makefile") {
```

```
61              isMakefile = true;
62              value = record.back();
63              record.pop_back();
64          }
65          if (value == "parameter") {
66              isParameter = true;
67              value = record.back();
68              record.pop_back();
69          }
70          if (value == "required") {
71              required = true;
72              value = record.back();
73              record.pop_back();
74          }
75          string parameter = record.back();
76          record.pop_back();
77
78          if (value == "random") {
79              randomConstraints.push_back(parameter);
80          } else if (required == true) {
81              requiredConstraints.insert(
82                  std::pair<string, string>(parameter, value));
83          } else {
84              if (isMakefile == true) {
85                  makefileConstraints.insert(
86                      std::pair<string, string>(parameter, value));
87              } else if (isParameter == true) {
88                  staticConstraints.insert(
89                      std::pair<string, string>(parameter, value));
90              } else {
91                  nonParameterConstraints.insert(
92                      std::pair<string, string>(parameter, value));
93              }
94          }
95      }
96
97      csvFile.close();
98
99      // Generate constraint-files
100
101     ofstream constraintFile;
102     ofstream makeFile;
103     char buffer[100];
104     int n = sprintf(buffer, "%d", (int)pow(2, randomConstraints.size()));
105     for (int count = 0; count < pow(2, randomConstraints.size()); count++) {
106         sprintf(buffer, "%d", count); //sprintf(buffer, "%.*d", n, count);
107         string cFileName = "config" + string(buffer) + ".tcl";
108         string fileLocation = "./constraintfiles/" + cFileName;
109         constraintFile.open(fileLocation.c_str());
110         constraintFile << "source " << argv[2] << "/legup.tcl\n\n"
111                        << "#####################################################"
112                           "##################\n"
113                        << "## Required Constraints:\n";
114         for (std::map<string, string>::iterator it =
115                  requiredConstraints.begin();
116              it != requiredConstraints.end(); ++it) {
117             constraintFile << "set_parameter " << it->first << " " << it->↩
    second
118                            << "\n";
119         }
120         constraintFile << "\n\n################################################"
121                           "######################\n"
```

```
122                         << "## Random Constraints:\n";
123
124         for (int offset = randomConstraints.size() - 1; offset >= 0; offset--) ↩
    {
125             constraintFile << "set_parameter " << randomConstraints[offset]
126                            << " " << ((count & (1 << offset)) >> offset)
127                            << "\n";
128         }
129         constraintFile << "\n\n################################################"
130                           "#####################\n"
131                        << "## Static Parameter Constraints:\n";
132         for (std::map<string, string>::iterator it = staticConstraints.begin();
133              it != staticConstraints.end(); ++it) {
134             constraintFile << "set_parameter " << it->first << " " << it->↩
    second
135                            << "\n";
136         }
137         constraintFile << "\n\n################################################"
138                           "#####################\n"
139                        << "## Static Non-parameter Constraints:\n";
140         for (std::map<string, string>::iterator it =
141                   nonParameterConstraints.begin();
142              it != nonParameterConstraints.end(); ++it) {
143             constraintFile << it->first << " " << it->second << "\n";
144         }
145         constraintFile.close();
146
147         // Generate Makefile for each constraint
148         string mFileName = "./makefiles/Makefile" + string(buffer);
149         makeFile.open(mFileName.c_str());
150
151         makeFile << "################################################################"
152                     "############\n"
153                  << "## Generated makefile:\n"
154                  << "NAME=" << argv[3] << "\n"
155                  << "LEVEL = " << argv[2] << "\n";
156         for (std::map<string, string>::iterator it =
157                   makefileConstraints.begin();
158              it != makefileConstraints.end(); ++it) {
159             makeFile << it->first << "=" << it->second << "\n";
160         }
161         makeFile << "LOCAL_CONFIG = -legup-config=" << cFileName << "\n"
162                  << "include $(LEVEL)/Makefile.common\n";
163         makeFile.close();
164     }
165     return (int) pow(2, randomConstraints.size());
166 }
```