NTNU – Trondheim
Norwegian University of
Science and Technology

High-Level Synthesis for hardware architectural exploration

Jørgen Holmefjord

Submission date: 24. September 2015
Responsible professor: Kjetil Svarstad, IET
Supervisor: Isael Diaz, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Contents

Acronyms

ANSI American National Standards Institute.

ASIC Application-Specific Integrated Circuit.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

GCC GNU Compiler Collection.

HDL Hardware Description Language.

HLL High-Level Language.

HLS High-Level Synthesis.

HW Hardware.

IR Intermediate Representation.

LLVM Low-Level Virtual Machine.

RTL Register-Transfer Level.

SoC System-on-Chip.

SW Software.

Chapter 1
Background

In the early days of digital hardware design, gates design and layout were performed manually by hand. With the rapid growth in the numbers of transistors per digital chip design, this method quickly became too timeconsuming and thus the need for new and more automated design methods rose. Register-Transfer Level (RTL) design using Hardware Description Language (HDL) has long been the standard in hardware design, but 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.

1.1 High-Level Synthesis

High-Level Synthesis (HLS) is not a new concept as it were introduced in research papers in the late 1970 and further researched and developed in the 1980 and early 1990s [5]. The available commercial HLS tools has however 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 is to use higher abstraction level, often a High-Level Language (HLL), to describe the functional specification of the circuit, and then let a tool help transform this specification into hardware represented as a RTL or HDL-model from the given target architectural models and design constraints. The typical HLS-flow is shown in figure 1.1 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.

Compilation

The first step of HLS is to compile the HLL into a formal model. This model can vary between different tools, and can be either a specific representation language or

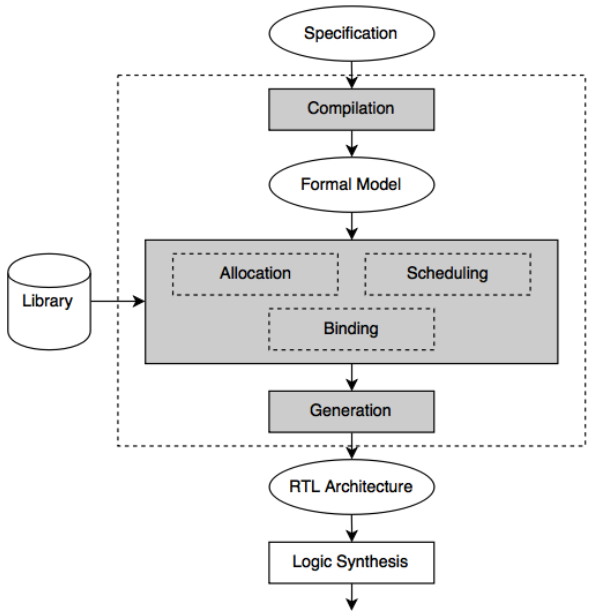

Figure 1.1: Information flow in a typical HLS tool (source: [3])

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 scheduling and bind task if found necessary 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 timeframe when they are 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) that sets 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. Depending on the intensiveness of the binding 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 is 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 the flexibility to optimize the unit binding to updates timing estimates and delays and loads given by the layout and floor-planning.

1.2 LegUp

The HLS tool used in this project is called LegUp [1]. 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 Field-Programmable Gate Array (FPGA) programming easier for software developers"*. LegUp takes ANSI C as input and generates synthesizable Verilog HDL as output. The developers of LegUp has 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 and possibility to configure the tool to generate generic Verilog to target other FPGA vendors or even Application-Specific Integrated Circuit (ASIC) through use of generic dividers, has been introduced. The big advantage of LegUp compared to similar, commercial tools is that it's open-source and thus can be configured to target different architectures, and the RTL and HDL generating part of the framework can be modified or replaced to fit the programmers needs. Since LegUp, in it's unmodified form, targets FPGA devices, it support three different synthesis flows; pure Software (SW), hybrid and pure Hardware (HW). The two first synthesis types will synthesize a TigerMIPS [6] soft processor, which will run the whole C file in pure SW flow and part of the C file in hybrid flow while the rest will be synthesized into hardware. The pure HW flow will synthesise the whole C file into hardware. It's the pure HW flow that will be the focus of this project.

Producing Verilog Output

LegUp is made up of two components; a frontend pass and a target backend pass to the Low-Level Virtual Machine (LLVM) compiler infrastructure. The information flow in LegUp, shown in figure 1.2, follows the same principle as the information flow described in section 1.1 The LegUp LLVM frontend takes LLVM Intermediate Representation (IR) compiled by clang 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 1.1. In the next step, RTL-module objects that represents the final hardware circuit are generated from each LLVM instruction. Ultimately, each RTL-module is printed to a file using the corresponding Verilog code for the given HW module.

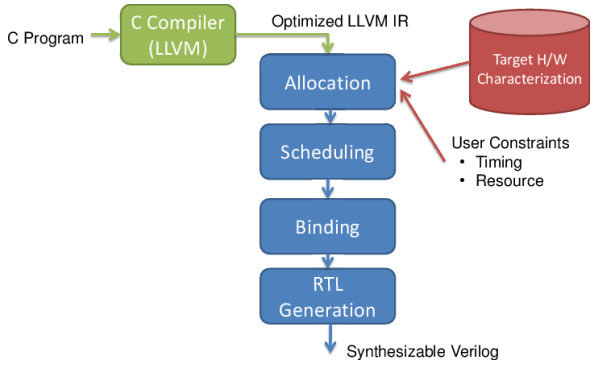

Figure 1.2: Information flow in LegUp (source: http://legup.eecg.utoronto.ca/docs/4.0/programmermanual.html)

1.3 LLVM

LLVM [4] 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 is widely used by for instance companies like Apple and Sony for iOS and PS4 development. LLVM support a large number of frontends for programming languages, including Clang [2] which support C, C++, Obcjective-C and Objective-C++ and is compatible with GCC. It also supports a large number of backend target architectures. Figure 1.3 shows how different source languages can be input to the LLVM compiler, which translate the source into an IR. The IR is then optimized using LLVMs optimizer. At this stage, different source languages can be linked together, and one can even link object files compiled using standard GNU Compiler Collection (GCC). The optimized IR is then translated into the target architecture by the backend.

Intermediate Representation

LLVM uses an human readable assembly-like, strongly typed RISC instruction set as their IR, with support for an infinite number of temporary registers of the form %0, %1, etc. LLVM IR can also output a dense bitcode format for serialization.

1.4 Power and area estimation

In order to compare different use-cases and result generated by LegUp, the area and power-usage will be estimated. Since this is not the main objective of this project, the automated area and power estimation tool-flow created by Joar Talstad in his

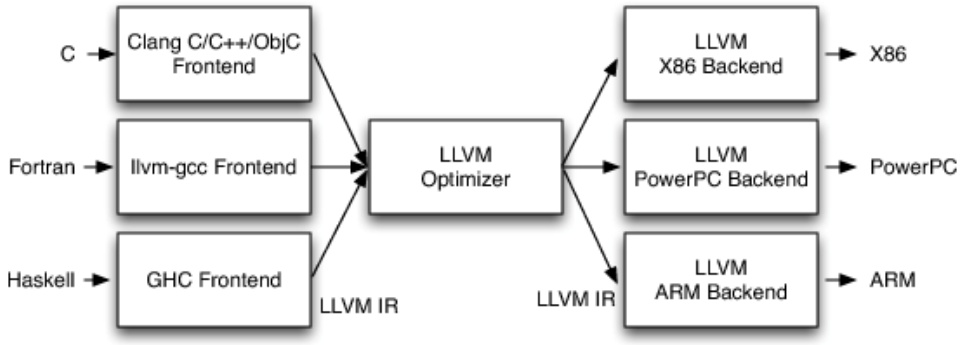

Figure 1.3: LLVMs three-phase compiler structure (source: http://www.aosabook.org/en/llvm.html)

specialization project [7] and master thesis [8] will be adapted for this purpose. The tool-flow he created uses Synopsys Primetime PX for power analysis and estimation.

1.5 Information and tool-flow

LegUp utilizes Makefiles to perform the HLS flow. The power and area estimation tool-flow described in section 1.4 also use Makefiles for its flow. It will thus be suitable to create a single Makefile that controls the information flow from the input of a C file to the output of Verilog files along with scorefiles on the area and power estimates.

References

[1] 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.

[2] clang.llvm.org. clang: a c language family frontend for llvm, 2007.

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

[4] 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.

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

[6] Simon Moore and Gregory Chadwick. The tiger "mips" processor, 2010.

[7] Joar Nikolai Talstad. Early stage power estimation and analysis on a multi-voltage design. Norwegian University of Science and Technology, December 2014. Specialization project report.

[8] Joar Nikolai Talstad. The title of the work. Master's thesis, Norwegian University of Science and Technology, 6 2015.