Bachelor's Thesis Computing Science

Radboud University Nijmegen

Review of HLS Toolchain

Bringing Flexibility and Performance to Software Engineers

Author:
Thomas Broekman
s1061435

First supervisor/assessor:
Prof. Sven-Bodo Scholz

Second assessor:
Dr. Peter Achten

August 29, 2023

Abstract

FPGAs are a type of hardware that allows developers to configure a circuit after runtime. FPGAs are more efficient than CPUs and have an easier and cheaper development process than ASICs. HLS tools take in a software language and compile it into a HDL, making FPGA development more efficient and open to software engineers. This document provides an introduction to and an overview of the HLS toolchain.

Contents

Chapter 1

Introduction

Nowadays, multiple types of hardware architectures are available. The most well known are the Central Processing Unit (CPU) and Graphical Processing Unit (GPU), which sit in almost every personal computer. Both the CPU and GPU work with a fixed instruction set, which limits application specific optimizations to the software domain. If developers want to execute one task very efficiently, they can develop an Application Specific Integrated Circuit (ASIC). ASICs are designed for a specific task, allowing for various application specific optimizations resulting in a very efficient design in terms of power usage, throughput and/or latency.

A disadvantage of ASICs is that the design cycle is very long and expensive. It is only feasible to develop an ASIC if the market for the final product is big enough. Another disadvantage is that ASICs cannot be changed after manufacturing, so any design flaws or missed features cannot be addressed until the next production batch.

Around 1980, Field Programmable Gate Arrays (FPGA), sometimes refereed to as User Programmable Gate Arrays, were designed to form a compromise between the generality of the CPU and the efficiency of the ASIC. FPGAs are hardware circuits that can be reconfigured after manufacturing, allowing for task specific hardware optimizations.

FPGAs are not as efficient as ASICs; a circuit on an FPGA is less efficient than the same circuit as an ASIC in terms of hardware needed, latency and power usage [35]. On the other hand, FPGAs often outperform CPUs because they allow developers to make use of optimizations such as parallelism and pipelining that are not always possible on a CPU. FPGAs have shown significant speedups in computation heavy tasks such as Microsoft Bing search [50], scanning Genomic DNA databases [38], textual pattern searching [22], and many other usecases such as Long Integer Arithmetic, RSA encryption, Molecular Biology, and Neural Networks [60].

An FPGA is configured by a bitstream that can be loaded after manufacturing. To obtain the desired bitstream, engineers provide a Hardware

Description Language (HDL) input to a process called synthesis, which turns the HDL into a bitstream. HDLs have a low level of abstraction, making development time consuming and error prone. Another problem with HDLs is that software engineers, who outnumber hardware engineers (e.g. by a factor 10 in the USA [6]), lack proficiency with HDLs.

Around 1990, academics began to develop High Level Synthesis (HLS) tools to combat these two problems. HLS takes in a software language such as C and has its output in HDL, raising the abstraction level for FPGA development and making FPGA development available to software engineers. These tools saw more widespread adoption around 2010, when major companies such as Intel [29], Xilinx [65] and Siemens [56] invested in and released their own HLS tools.

There is a lot of good work available on HLS, but the barrier of entry for a novice is still very high, especially for software engineers. This document tries to fill that gap by providing an introduction to and an overview of the HLS toolchain. After reading this document, the reader will have enough background knowledge to start developing HLS based applications and/or start research in the HLS toolchain.

This document discusses FPGAs in chapter 2. Chapter 3 discusses how to configure FPGAs and introduces HLS, chapter 4 discusses debugging in the context of HLS and chapter 5 discusses remaining development steps such as verification, testing and performance logging. In chapter 6 scientific HLS based applications are discussed, in order to see what HLS is used for and which tools are used. This document ends with with a discussion about future work and a conclusion.

Chapter 2

Field Programmable Gate Arrays

This chapter discusses the hardware that HLS based applications run on: FPGAs. The first section is a quick refresher on the very basics of a hardware circuit. The second section discusses how an FPGA works and the third section discusses how an FPGA can fulfill an accelerator role. Finally, in section four briefly illustrates how an FPGA can outperform a CPU on both execution time and energy consumption.

2.1 Hardware Basics

At the lowest logical level, there are bits (0 or 1). Multiple bits can be used together to represent objects like an integer or a string. By combining basic operations such as NEGATE, AND, OR and XOR, small components can be formed such as a full adder, a circuit that takes in a carry bit and two bits and returns the sum of those and their carry. Bigger components can be build by combining smaller ones. For example, by combining 32 full adders one creates 32 bit integer addition. In this manner, arbitrary computations can be made.

To store state, flip-flops and latches are used. These are circuits that can save a single bit, by combining them one can store multiple bits and thus arbitrary data structures. A clock, emitting a signal alternating between 0 and 1 on a given frequency, is used to indicate when the state should be updated. By negating the clock signal, the flip-flop can be switched to store on rising edge or falling edge.

However, how fast the state updates is not only determined by clock frequency. Instead, clock frequency needs to be limited so that actual electronic signals (e.g. 0-1 volt for 0 and 3-5 volt for 1) have time to stabilize. The longest path delay between any two storage units will determine how long the circuit needs to stabilize. This path is called the critical path. It

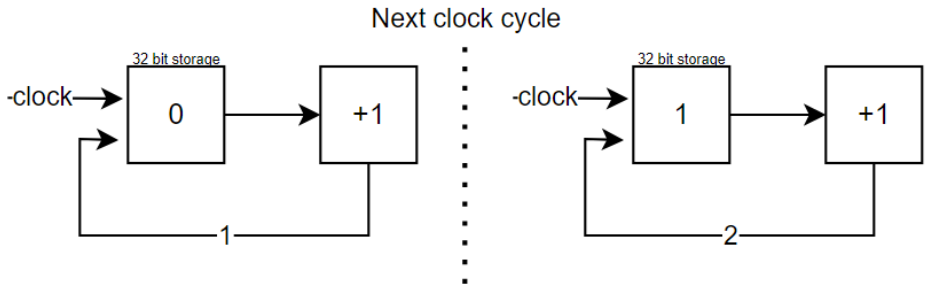


Figure 2.1: Counter circuit.

can be beneficial to break up critical path computation into multiple clock cycles, in order to allow a higher clock frequency.

As an example consider the circuit in figure 2.1. The 32-bit storage component consists of 32 flip-flops. The addition by one component consists of 32 full adders chained together, having as input the output of the 32-bit storage and a fixed value 1. Every clock cycle (e.g. at the rising edge) the flip-flops store their new value and the next input value will be incremented by one. The critical path is the path going from the output of the storage to the input. Say this takes σ seconds to stabilize, then the frequency of the clock must be lower than $\frac{1}{\sigma}$ in order to guarantee that the counter is stable.

Lastly, some circuits have multiple clock domains with a different clock frequency. It is important that signals from different clock domains do not interfere, as this would cause instability in the circuit. This is known as clock domain crossing.

2.2 FPGA Architecture

As mentioned in the introduction, an FPGA is configured by a bitstream. This bitstream is stored in non-volatile memory, so that the PFGA can be turned on and off without losing its configuration. To accommodate for the configurability needed by developers, a typical FPGA consists of three components, as can be seen in figure 2.2. This three component architecture is referred to as the Logic Cell Array (CLA) architecture.

1. An array of configurable logic blocks (CLB)
2. Interconnection resources
3. A perimeter of I/O blocks

A CLB provides a small configurable functionality. Once configured, it could check if four bits are all zero or add two two-bit integers together.

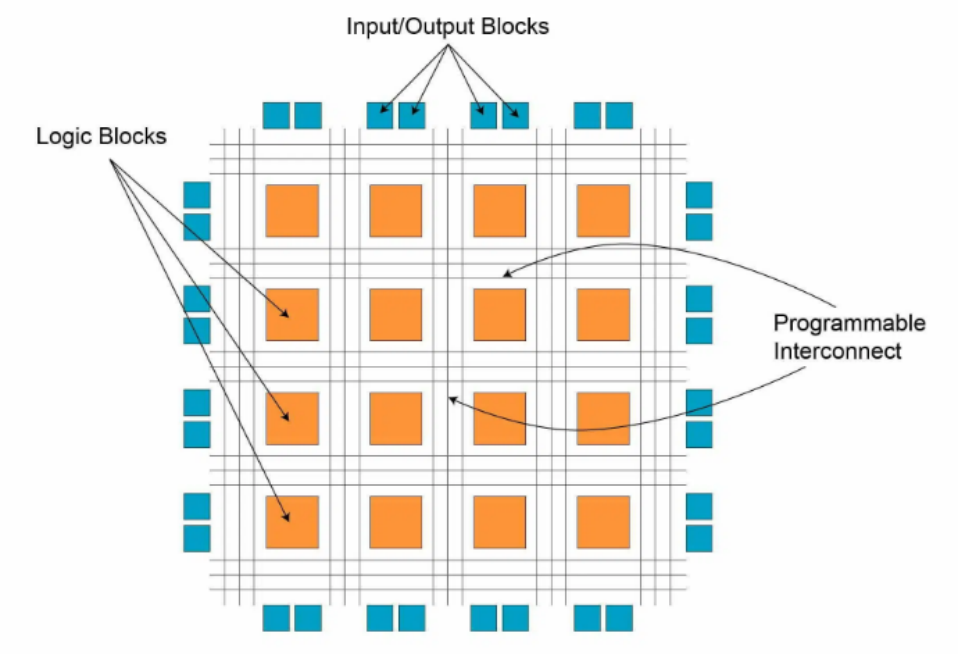


Figure 2.2: FPGA Logic Block Grid.

The interconnect resources have two purposes. Firstly, they allow the developer to combine multiple CLBs into bigger components. Secondly, they allow the developer to connect the circuit to I/O blocks. As it is not feasible to connect every CLB to every other CLB, a typical FPGA will have a hierarchical interconnect, with the lowest level tightly connecting neighbouring CLBs and the highest level consisting of across device wires. This makes sense because neighbouring CLBs will often form a single component, which can then communicate via higher level wires to form a complete circuit and connect to I/O blocks.

This hierarchical networking is not reserved to FGPAs and can also be found in other configurable architectures such as the MATRIX architecture [41]. A benefit of tightly connecting neighbouring CLBs is providing a minimal network delay between CLBs that use each other in a single computational component, thus decreasing the delay of the critical path.

The I/O blocks allow communication between the FPGA and other parts of the system. A use case could be reading two big arrays from input pins, computing their inner product and outputting it back to output pins.

2.2.1 XC3000 FPGA Series

To get a better understanding of how an FPGA works, we look at a concrete architecture in more detail. The XC3000 FPGA architecture [25] has been

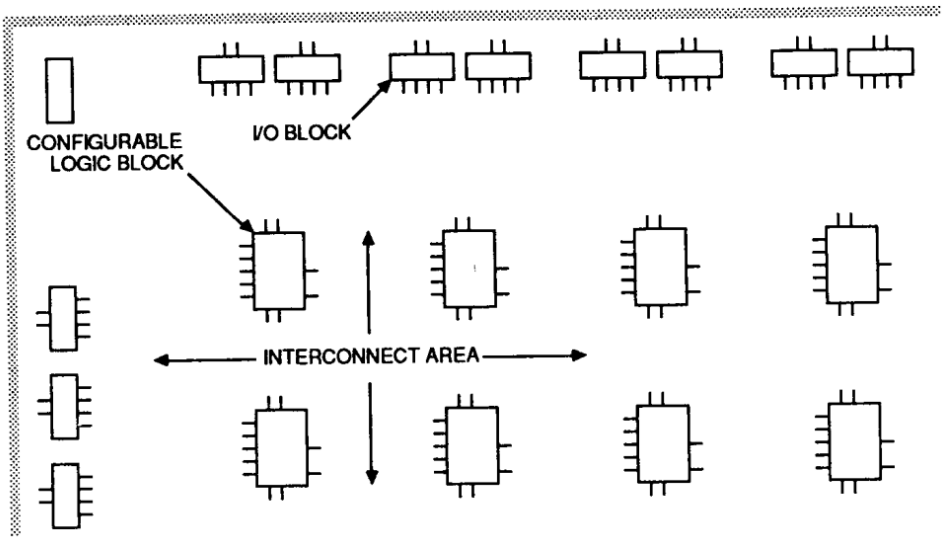


Figure 2.3: XC3000 FPGA CLA architecture [25].

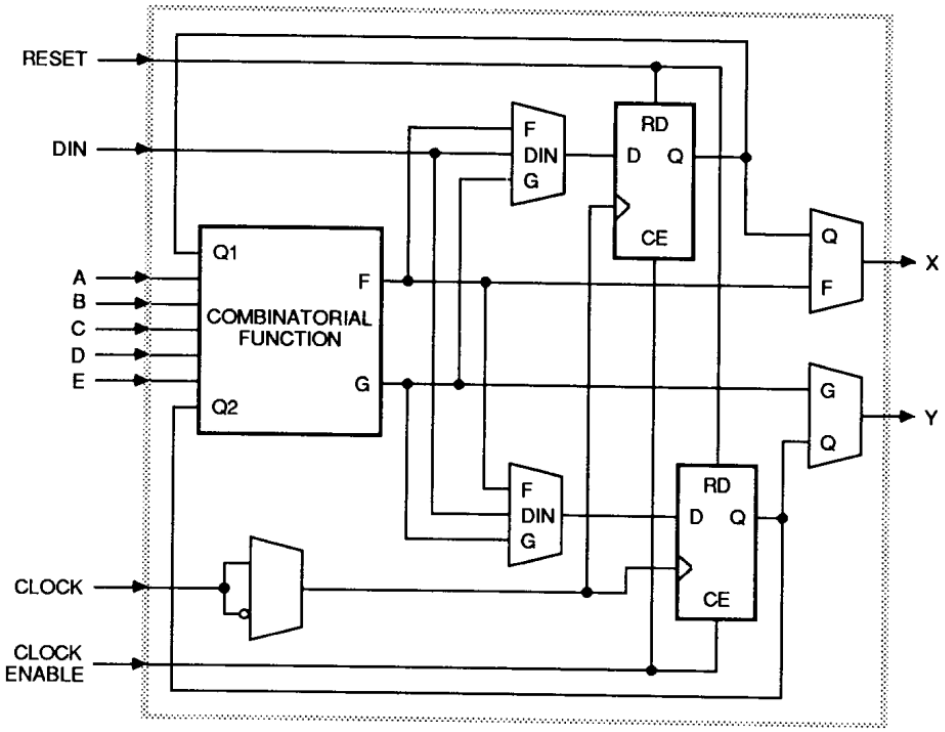


Figure 2.4: XC3000 Configurable Logic Block [25].

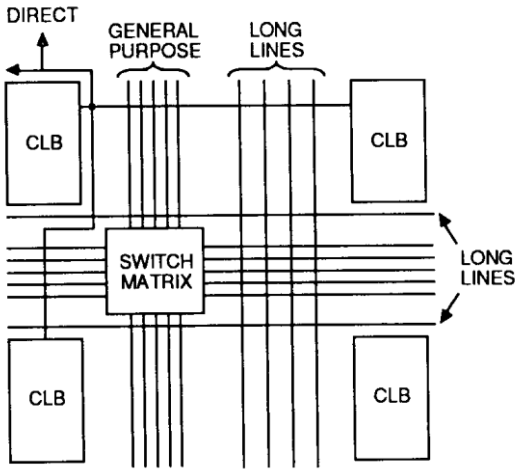


Figure 2.5: XC3000 Routing Resources [25].

around since 1988 and a high level overview of the architecture can be seen in figure 2.3. One can clearly recognize the three components of the CLA architecture.

The XC3000 CLB is depicted in figure 2.4. The combinatorial function block can implement any five input boolean function, or two independent functions up to four variables each. The output of those functions can then be stored in the two flip-flops. There is a possibility for a feedback loop, with inputs Q1 and Q2. The output of the CLB can come from the flip-flop, but if the developer needs the signal before the next clock cycle, the CLB can also be configured to have the output come directly from the combinatorial circuit. This choice is provided by the LUTs connected to the X and Y output. The LUT right after the clock input enables the CLB to be configured for either rising or falling edge triggered flip-flops. Note that in this picture the LUTs do not have choice input bits, because these come from the bitstream that configures the FPGA.

The routing resources of the XC3000 architecture can be seen in figure 2.5. There are three types of routing resources, clearly representing the hierarchical network we describe in the beginning of this section.

1. Direct connect. Direct connect wires run directly from the output of a CLB to its neighbouring CLB's input.

2. General purpose lines. In the FPGA there is a grid of 5-bit lines that have a switch matrix at every intersection. A switch matrix connects in and outputs and allows for arbitrary paths from one CLB to another.

3. Long lines. Long lines run the entire width or height of the device. As opposed to general purpose lines, there is no switch matrix in between

and the signal is available to all CLB and I/O blocks in the given row or column.

Finally the XC3000 I/O blocks each control a single pin. The options for the I/O block (e.g. voltage thresholds) lie too close to electrical engineering for this document and the interested reader is referred to page 2, paragraph 6 of [25].

2.3 FPGA + CPU systems

In the previous sections, we have viewed the FPGA as a computing device on its own. Similar to GPUs, FPGAs nowadays are also be used as accelerators. Heterogeneous systems containing an accelerator FPGA usually have the CPU in a leading role.

To use an FPGA and a CPU in this way, the CPU needs to be able to send and read data to and from the FPGA, which can be done by configuring the FPGA's I/O blocks correctly. Secondly, the CPU might need to configure the FPGA at runtime. This can be done by embedding the bitstream configurations into the program that runs on the CPU and then having the CPU send this to the FPGA when needed. Loading a new FPGA configuration at runtime is only worth it if the time saved is greater than the configuration time of the FPGA. This could also be estimated at runtime e.g. based on the size of the dataset.

2.3.1 Garp

An example of such a heterogeneous system is the the Garp architecture [23]. It uses a MIPS CPU and a reconfigurable coprocessor (in our case, this is any FPGA). The design flow of Garp can be seen in figure 2.6.

A hardware description is passed through a synthesizer to obtain a bitstream in .config files. The function of a synthesizer is equivalent to that of a compiler for hardware circuits and is described in more detail in section 3.1. Suppose we have a configuration named "add", then the corresponding bitstream can easily be embedded into the C program by the following code:

```
char config[] =
#include "add3.config"
;
```

The standard C preprocessor will put the bitstream bytes in the config array. In order to load the configuration at runtime, the C compiler would need to be extended with a statement statements for that load configurations from a CPU onto an FPGA. Assuming that this instruction is present, the developer can compile the code per usual and obtain an executable that can be run on a CPU which uses an FPGA as coprocessor.

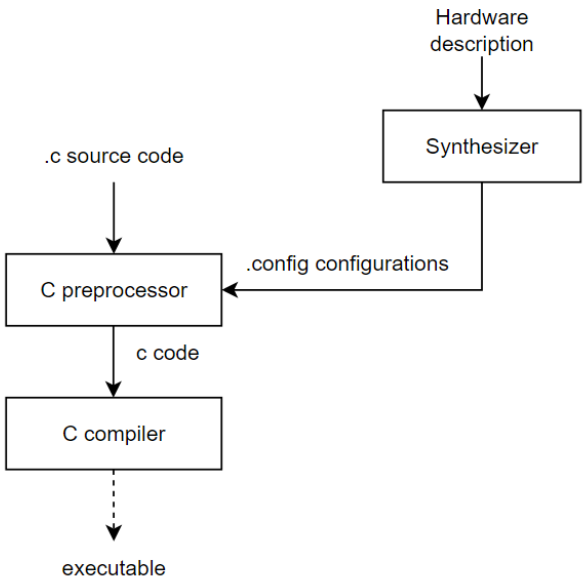


Figure 2.6: Basic heterogeneous (CPU, FPGA coprocessor) programming environment.

The Garp architecture [23] was not manufactured at the time of writing the paper. To evaluate the heterogeneous computing system, GARP is tested in a simulator. The speed ups are reported in table 2.1. Indeed we observe significant speedups for computation heavy tasks. For the sorting benchmark a memory usage optimized merge sort is implemented, but the 2.1 speedup is still a lot less than other speedups. Although the authors do not reflect on this, we think this is because sorting does not utilize the pipeline computation that FPGAs bring to the table as effectively as other benchmarks such as DES encryption. Another reasons could be that the sorting benchmark is dominated by memory accesses.

Bench mark	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt of 1 MB	3.6s	0.15s	24
Dither of 640 by 480 pixel image	160ms	17ms	9.4
Sort of 1 million records	1.44s	0.67s	2.1

Table 2.1: Benchmark results comparing SPARC (simulated) vs Garp [23].

2.4 FPGA strengths

Now that we know what an FPGA is capable of, we can take a look at how it improves performance in terms of execution time and energy consumption when compared to a CPU. We do this by providing code samples compiled via Godbolt Compiler Explorer [15] and comparing them to corresponding handcrafted FPGA circuits. We are using the x86-64 gcc 13.1 C++ compiler with the optimization flag -O3 passed to it.

The first major strength of the FPGA is parallelism. To illustrate this example we work with optimization level -O2, as the -O3 assembly output is not human readable. Consider computing the inner product of a 10 dimensional vector v: $\sum_{i=1}^{10} v_i^2$, see figure 2.7. As we can see on line 4, 9 and 10 of the assembly, the C code loops through the whole array and then executes the multiplication to obtain the squares one by one. The circuit on the other hand can execute as many squares as there are physically available at the same time.

Even if the length of the array was not given in advance, the FPGA circuit could make e.g. 16 square components next to each other and stream the array 16 by 16 elements at a time, still outperforming the sequential approach of the CPU.

Parallelism can also be useful in other areas such as branching. Consider figure 2.8, which contains a simple function which calculates $x * (y + 4)$ if b is true and $y * (x + 4)$ otherwise. As we can see, the assembly compares the two and then jumps to the corresponding calculation. Modern CPUs will have a pipelined execution preparing one of the two branches in advance. In case of a missed branch prediction, the CPU will calculate execute the other branch. This incurs a significant delay as essentially the entire branch needs to be executed from the start. On the other hand, the FPGA circuit will simply calculate both sums in parallel and select the correct one via a look up table with the boolean value as choice bit.

Another major strength of the FPGA is pipelining (or operation chaining). As discussed in section 2.2.1, the output of a CLB can be directly passed on to the next, rather than being saved before being used again. This means that, depending on the delay and the clock frequency, calculations that would take multiple cycles in a CPU will only take a single cycle in the FPGA. Consider figure 2.9, which considers a program multiplying four numbers. As we can see, there are three multiplications in the assembly code. This means that at least three cycles are needed on the CPU (assuming a single cycle multiplier is available), where we disregard the move operation on line 4. While on the FPGA this could be done it at most two, arguably one cycle.

To allow for longer circuit delays, FPGAs generally use a shorter clock speed (10-250 MHz) compared to CPUs (1-4 GHz). Because of parallelism and pipelining, an FPGA will get more work done in a single cycle. Hence

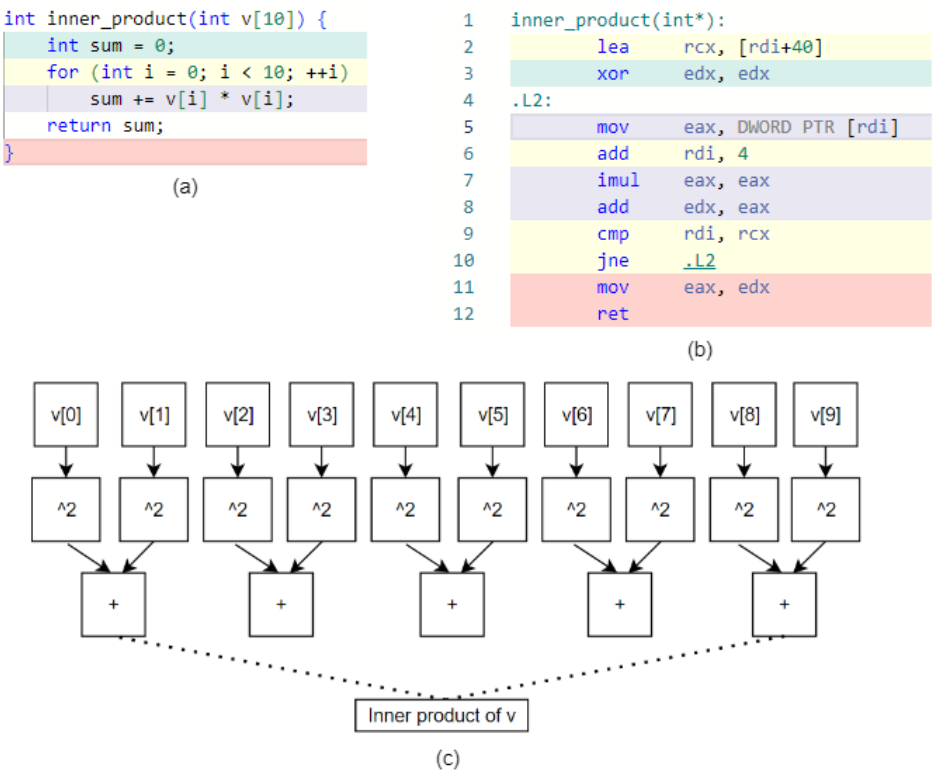


Figure 2.7: (a) C++ code, (b) assembly code (-O2) and (c) circuit for computing the inner product of a 10 dimensional vector.

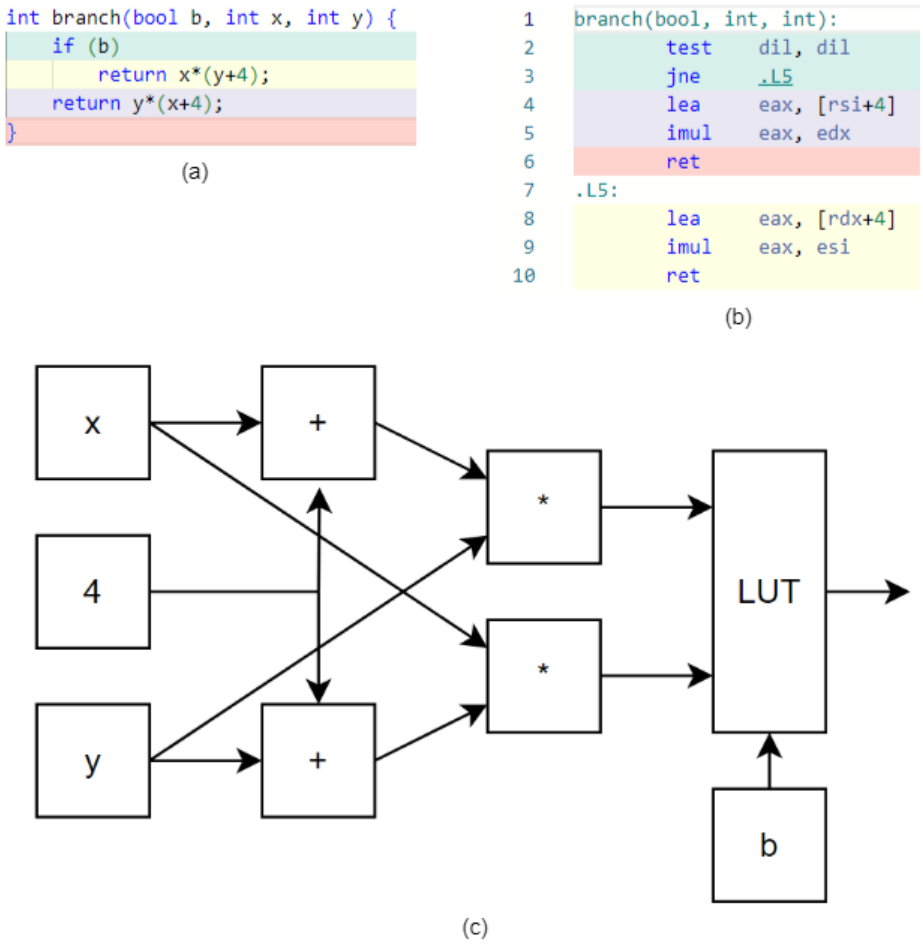


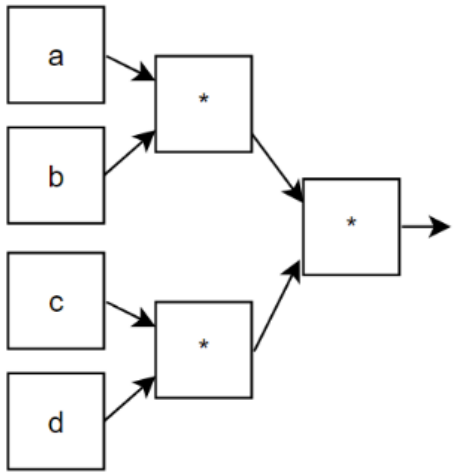
Figure 2.8: (a) C++ code, (b) assembly code (-O3) and (c) circuit for a branched calculation.

```
int sum(int a, int b, int c, int d) {
    return a * b * c * d;
}
```

(a)

```
1   sum(int, int, int, int):
2           imul    edi, esi
3           imul    edi, edx
4           mov     eax, edi
5           imul    eax, ecx
6           ret
```

(b)



(c)

Figure 2.9: (a) C++ code, (b) assembly code (-O3) and (c) circuit for multiplying four numbers.

FPGAs can still outperform CPUs in terms of throughput and latency.

The more efficient operations partially explain the energy efficiency that an FPGA brings. Another factor is that an FPGA does not need to use unnecessary bits. For example if a program uses a single byte counter, a 64bit architecture still has 64bit registers. An FPGA will only use eight bits, thus saving energy on 56 non-working bits.

Finally, there are operations for which the FPGA is simply better equipped than a CPU. For example, bit manipulation operations can be executed much more efficiently on hardware than in software, which explains why FPGAs perform so well on cryptographic algorithms. Pipelining and parallelism enable the FPGA to perform well in areas such as big data, networking and machine learning. In general, the combined strengths make FGPAs an excellent choice of hardware for implementing computation intensive (streaming) applications.

Chapter 3

High Level Synthesis

As described in chapter 2, an FPGA is configured via a bitstream. In this chapter we take a look at how a functional bitstream can be obtained. This is normally done through compilers that operate on different levels. We distinguish three levels.

1. Low level hardware description, directly describing a hardware circuit using Verilog or HDL.
2. High level hardware description, directly describing hardware using more abstract packages.
3. High level synthesis, using a compiler that turns software source code into a hardware circuit.

In section 1 we describe how hardware description works. In section 2 we discuss high level HDLs and personal experience with the high level HDL Clash [51]. We end this chapter with section 3, where we discuss high level synthesis.

3.1 Hardware Description Languages

As we mention at the start of this chapter, FPGAs are configured via a bitstream. The bitstream describes the actual configuration for a specific FPGA architecture. But HDLs are the same for all architectures. Similar to the GCC compiler translating the standardized C code to a specific instruction set, HDL compilers compile code into bitstreams via the following steps:

1. The circuit functionality is described in HDL.
2. Synthesis: The HDL code is turned into a netlist which describes the logical connections and components of the design.

3. Mapping: Depending on the specific FPGA design, the netlist is mapped to specific resources (e.g. LUTs and flips flops).

4. Place and route: The design is placed onto available CLBs and routing resources. This step also optimizes for timing and checks if the required clock speed is feasible (as mentioned in chapter 2, this depends on the critical path).

5. Finally the bitstream is obtained and this can be loaded onto a specific FPGA design.

Generally the HDL description is provided by an engineer or a compiler. Similar to software languages, many HDLs exist. The most well known are VHDL [47] and Verilog [46]. HDLs are much like software languages in the way the code connects inputs to outputs. As an example, consider the the following VHDL code which describes the full adder circuit in figure 3.1. A full adder circuit is a circuit that can be used inside an integer addition component. In case of a result greater than 1, a carry is generated and routed to the CarryOut signal. The the next full adder gets this signal as an input in CarryIn. Note that there is not a one to one translation between the XOR, OR and AND operations in the code and the logic gates in the physical circuit. This is due to optimizations that the VHDL compiler is allowed to make.

```
entity full_adder_vhdl_code is
 Port ( A : in STD_LOGIC;
 B : in STD_LOGIC;
 CarryIn : in STD_LOGIC;
 Result : out STD_LOGIC;
 CarryOut : out STD_LOGIC);
end full_adder_vhdl_code;

architecture gate_level of full_adder_vhdl_code is

begin
 Result <= A XOR B XOR CarryIn ;
 CarryOut <= (A AND B) OR (CarryIn AND A) OR (CarryIn AND B) ;
end gate_level;
```

With knowledge of the internals of a full adder circuit, someone who understands software languages can certainly understand the VHDL code extract. However, a software engineer is generally not equipped to work with HDLs effectively. This is because of the differences between software languages and HDLs. We consider there to be three main differences between hardware and software languages, besides the fact that HDLs are made to

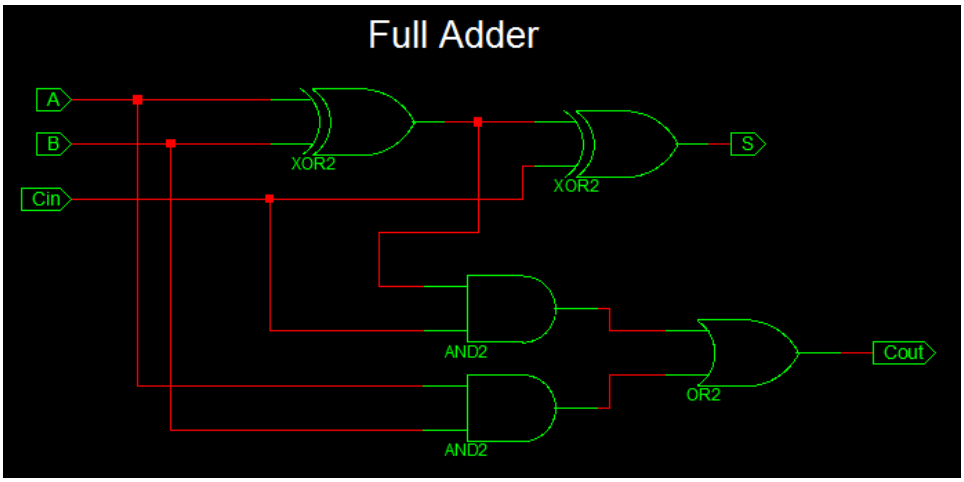


Figure 3.1: Full Adder Circuit.

design circuits and software languages are designed to be executed on a CPU.

Firstly, HDLs are on a very low abstraction level and are close to the hardware. Even though sophisticated design verification tools exist, this low abstraction level makes it challenging to design more complex systems. Secondly, HDLs inherently support concurrent execution. On a circuit multiple components operate in parallel and this requires a different way of thinking than software languages that are single threaded by default such as C or Python. Thirdly, HDLs have an explicit notion of the circuit clock in them and hardware designers have to optimize for clock speed and the critical path, whereas software engineers usually focus on leveraging sophisticated algorithms to minimize the amount of operations needed.

3.2 High Level HDL

To counter this low level nature of HDLs, High Level HDLs have been made. A well known example of this is SystemC [28], a hardware description languages that allows description on a cycle-to-cycle level, but also on a more abstract level using C++ features such as classes. SystemC is mostly used for complex systems and heterogeneous systems such as described in section 2.3. Another example of such a high level HDL is Clash. Clash is an open source programming language that borrows its syntax and semantics from the functional programming language Haskell and compiles source code into VHDL or Verilog.

I (Thomas), together with a group of other students, have designed hardware components to receive/send packets and verify/generate their checksum on the link layer for FPGAs using the high level language Clash for

the course Software Engineering (NWI-IBI001). Because the whole group consisted of software students, the problems we encountered fit very well inside this thesis. The three differences mentioned in the previous section turned out to be problematic for our productivity.

The low abstraction level came to us in the form of dealing with explicit hardware components that were hard to understand with our limited background knowledge. For example, we had to explicitly think about a hardware component that turns two 4-bit signals (one on the rising edge and one on the falling edge of the clock) into a single 8-bit signal. Whereas in software we would most likely get our input from a pointer or a stream.

The concurrent execution made us choose for a design where everything was in a byte by byte stream. The Clash languages facilitated this very well, but because of this we did have to optimize the code so that the critical path was short enough for the clock speed to be on the same frequency as the Ethernet.

Lastly, the explicit notion of clock domains (recall section 2.1) was a completely new concept which required some time to get used to.

We conclude that even though High Level HDLs exists, they are still hard for software engineers to understand and use effectively. The main problem is that the conceptual level is still at a hardware level. Software engineers generally lack the knowledge and expertise to understand this without extra education.

3.3 High Level Synthesis

Throughout the previous two sections, we have described (High Level) HDLs and the challenges they bring. We have concluded that most software engineers, unless educated, are unable to use HDLs effectively. High Level Synthesis (HLS) compilers are created to solve exactly that problem. HLS compilers aim to bridge the gap between software and hardware design by allowing software engineers to describe the desired behavior in a software language and generating a corresponding hardware implementation. This is most often done by taking in (a subset of) C as source code and generating a HDL description, which is then synthesized into a bitstream for a specific FPGA design, as described in section 3.1.

HLS Compilers are relatively new, with the earliest dating from 1998, but already more than fourty HLS Compilers have been developed. A list of HLS compilers can be found in Table 1 of [44]. Although Wikipedia is far from a scientific source, it is worth to notice that their High Level Synthesis page reports even more compilers [64].

There exists both academic and commercial HLS compilers. The most well known academic compilers are Bambu [49], Dwarv [43] and LegUp [6]. The most well known commercial compilers come from companies (some-

times FPGA vendors) such as Intel HLS [29], Xilinx' Vitis HLS [65] (which replaced Vivado HLS [66]) and Siemens' Catapult HLS [56]. In the rest of this section we first go over HLS compiler characteristics by looking at the three academic compilers and then briefly discuss advantages and disadvantages of using academic compilers compared to commercial compilers. We finish this section with some pointers to performance and optimization.

3.3.1 Academic HLS Compilers

A typical HLS Compiler takes in (a subset of) an already existing software language. Bambu, Dwarv and LegUp are all based on the C programming language. However, FPGAs come with limitations so not all software concepts can be conveniently implemented. For example, there is no stack on an FPGA and implementing one would limit parallelism, which is one of the strongest features of FPGA design [43]. This means that function calls cannot be recursive and that there is no non-const static data. Dynamic memory allocation is also unsupported for C hardware synthesis. The choice for the C language also means that object oriented programming is not supported.

On the other hand, the HLS compilers accept a broad range of C concepts such as unions, function calls, (do-)while loops, return and break statements and data types such as floats, multi-dimensional arrays, unions and structs.

Next to standard C, the user of a HLS compiler needs to annotate the source code. These annotations are used to indicate which HLS optimizations should be applied on what parts of the code. For example, LegUp supports Pthreads and OpenMP pragmas, and parallel threads are synthesized into parallel-operating hardware [44]. In the case of a heterogeneous setup, annotation becomes even more important because it tells the compilers which parts should be executed in hardware and which part should be on the processor.

An example of a HLS compiler targetting heterogeneous systems is LegUp. It specifically targets a system with a 32-bit MIPS soft processor and an FPGA. This is ideal because inherently sequential operations such as iterating over a list is well-suited for software execution, while inherently parallel operations are better suited for hardware execution. LegUp helps the programmer choose by profiling the execution and then suggesting program segments, as can be seen on step 3. of figure 3.2. Once a segment is chosen to be executed on the FPGA, the function call in the C code is replaced by a function wrapper responsible for communication with the FPGA: sending the arguments and receiving the results.

After the source code is provided, the HLS compilers leverages an already existing compiler to compile and optimize the source code. Bambu uses the GCC compiler, Dwarv uses the CoSy CFront compiler, but LegUp does not mention any specific compiler. An advantage of using compilers such as GCC is that they are robust and allow the user to enable errors, warnings

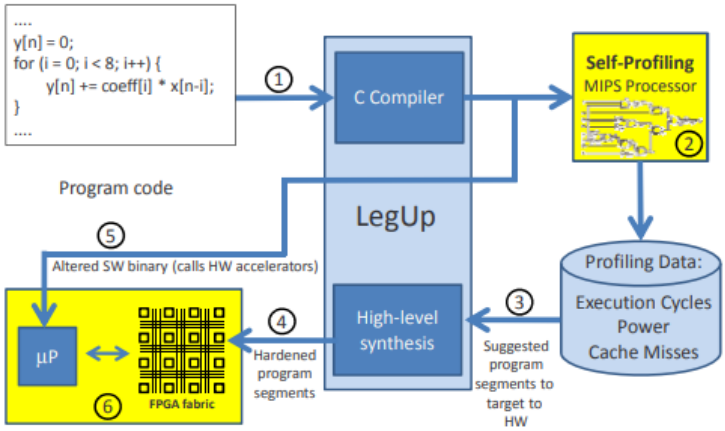


Figure 3.2: Design flow with LegUp [6].

and optimization flags as they normally would.

The resulting abstract syntax tree is used to generate the hardware description. For general use HLS compilers, modules are generated by producing a data path, a finite state machine and a memory interface. The internals of this process are out of the scope of this document. The interested reader is referred to the pointers in section III.A of [43] and section 4.1 of [6].

Finally, the obtained HDL can be combined with other hardware modules or synthesized onto an FPGA directly.

Many HLS compilers verify their own generated circuitry based on the source code, this is called cosimulation and we discuss this in depth in chapter 5. Cosimulation is done by compiling the source code and simulating the circuitry. Given the same inputs, the outputs are compared and if they do not match there is an error in the generated circuitry. This method does not catch design flaws and should not be mistaken for functionality testing.

All three academic compilers use already existing tools to build their HLS compiler on. An example of this is the use of already existing C compilers. Another example is how support floating point operations is implemented: Dwarv uses Xilinx's tool coregen and Bambu uses the FloPoCo library.

A special feature of academic compilers is that they are open source and designed in a modular way so that new features can be easily added. This helps further research in HLS compilers and other HLS area's such as debugging. Because of this, proof of concept implementations are often based on an already existing academic compiler.

3.3.2 Commercial HLS Compilers

As we can see in [44], academic compilers were the majority in 1998-2006. Most of these compilers are now abandoned, but the success of those compilers generated the interest for commercial HLS compilers. These have a similar workflow to academic HLS compilers and do not necessarily produce better results in terms of performance. However, commercial compilers are more robust and support more features than academic compilers. Examples of such features are allowing multiple input languages, multiple output languages and customization of kernels for memory bank usage, interfaces and throughput.

Another benefit of commercial compilers is the resources around the compiler. Their documentation is of high quality and extensive manuals are often freely available via sites of the vendor. Moreover, companies often invest in teaching modules on how to use their tools, so that engineers have an easier time using it. The Intel HLS Compiler [29] is a great example of this.

The Intel HLS Compiler takes in C++ and converts it to a circuit description. The generated circuit can be automatically simulated in ModelSim [57] to verify circuit functionality based on the source code. The Intel HLS Compiler comes with its own IDE. To get engineers to used to their FPGA related products, Intel provides high quality public tutorials which can be found on their YouTube channel Intel FPGA.

Specifically for HLS, there is a 7 part tutorial series which can be found on YouTube by searching for the following names:

1. Introduction to High-Level Synthesis (Part 1 of 7)
2. HLS Interfaces (Part 2 of 7)
3. HLS Loop Optimizations (Part 3 of 7)
4. HLS Data Types (Part 4 of 7)
5. HLS Local Memory Optimizations (Part 5 of 7)
6. HLS Performance Optimization (Part 6 of 7)
7. HLS Optimization Example: Matrix Decomposition (Part 7 of 7)

Intel also provides free exercises per tutorial, which can be downloaded via the following url:
`https://www.intel.com/content/www/us/en/programmable/customertraining/Videos/HLSPartX.zip`, where X should be replaced by a number 1 to 7.

Other academic compilers also provide these types of resources and links on their site or a quick google search will direct you to them.

A downside of commercial tools is their tendency to form a closed system. For example, the Intel HLS Compiler is inside the Intel Quartus Prime Design Software package and the Intel HLS Compiler has extra optimizations for Intel FPGAs. Another downside is that they charge for their product. Even if a product is free, it is often limited and the developer needs to pay for a better experience.

3.3.3 Performance and Optimizations

As HLS tools are often used to get better performance in terms of execution time or power usage, optimizations are of great interest. The first layer of optimizations is those made by the software compiler (e.g. GCC). Then come the HLS specific optimizations focus on using FPGA features (see section 2.4) such as spatial parallelism, operation chaining and minimizing the amount of resources used. For an overview of these optimizations, see section III of [44]. Some compilers also leverage domain specific optimizations, see the domain column of table 1 of [44]. Commercial compilers made by FPGA vendors also support FPGA specific optimizations, as with the Intel HLS compiler.

In case of heterogeneous systems, performance is also affected by deciding which part of execution are on the FPGA and which parts on the CPU. However, this is mostly up to the engineer to decide.

Chapter 4

Debugging

When debugging an HLS system one can either simulate or observe the physical world, and one can either look at the software code or at the hardware circuit. When we combine these two choices, we get a total of four possibilities.

(1) Firstly, a developer can take the source code and debug it with CPU software development tools. For example, a developer can take the C source code and use GDB [14] to check for correctness of the source code. (2) Secondly, the developer can compile the source code with an HLS compiler and simulate the resulting circuit to see how it behaves. (3) Alternatively, the circuit can be loaded onto hardware and the developer can observe the hardware in execution. (4) Lastly, the hardware execution is observed, but the signals are linked back to the source code to give the developer insight into execution on a software level. The circuit is altered to obtain visibility into the needed signals.

Each method has benefits and drawbacks and we go over all four methods individually. We end this chapter by taking a look at existing HLS debugging tools.

4.1 Simulating the source code on a CPU

As we mention in chapter 3, HLS compilers usually take in (a subset of) an already existing programming language, such as C. In the rest of this chapter we will take C as our example source code language, but this can really be replaced with any source code programming language that is accepted by any HLS compiler.

When one trusts the correctness of HLS compilers, verifying our C code functionality implies verifying the functionality of the generated circuitry. Therefore a developer can debug their HLS project by loading C source code into an already existing debugger and using the software debugging skills that they already possess. This way, the developer gains insight in

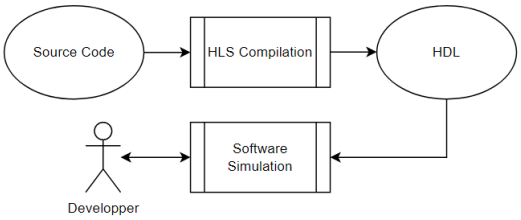


Figure 4.1: Graphical representation of software emulation

the runtime behavior of their code and they can simulate in- and output themselves. A benefit of this approach is that software developers are already used to this kind of debugging, and mature software debugging tools are readily available.

A widely used debugger for the C programming language is GDB. GDB has a very useful interface and is easily understood by software developers. Its main features are inspecting and modifying variables during runtime execution and inserting breakpoints at lines in the source code, so that the program execution stops whenever the program reaches that line. This allows developers to gain insight into execution at critical points. GDB also features step by step execution, and the ability to isolate function calls and test them independently.

However, debugging tools like GDB are not designed for debugging circuits. And this approach will not give developers real insight into what is happening inside the generated circuit. This is especially important when an FPGA interacts with other components such as untrusted networks, or when the FPGA acts as an accelerator, such as in the garp architecture discussed in 2. In these settings, developers need information about the data that is being passed through the FPGA circuit and how that data is being processed.

4.2 Simulating the circuit

To get insight into the hardware execution, this approach simulates the actual circuit, instead of the source code. The developer uses circuit simulation software such as SYCL [30], Quartus Prime [31] or Isim [2] to analyze the circuit, in our case a circuit generated by a HLS compiler. See figure 4.1 for a graphical representation of this process.

Features of software simulation are in principle similar to the GDB features that are discussed in the previous section, but translated to circuits. Developers can pause execution and inspect/modify the state of hardware units.

Another benefit of software simulation is that, as with approach 1, we do

not need to have the actual hardware available in order to make progress. When working with multiple developers on limited amounts of hardware devices, or when working with developers that work from a distance, this becomes a point that is worthwhile to consider. Long FPGA synthesis times, ranging from minutes to hours [10], encourage developers to work with software simulation, where the circuit can be changed/updated in a much shorter time. However, it should be mentioned that simulations are slower than actual hardware, therefore reaching bugs that only occur after billions of cycles can take significantly more time in a simulation [21].

Another disadvantage of simulation is that there are bugs which occur in the physical setting, yet are not observable in a simulation. These bugs may be caused due to interactions with the environment, interfaces with legacy IP blocks, race conditions or only occur after long running times [33]. Simulation is not sufficient to discover these type of bugs. Developers will only encounter such bugs after observing unexpected behavior when running the HLS generated circuit on an FPGA in a production setting. This brings us to approach 3: observing the circuit in a production setting.

4.3 Observing the circuit

In this debugging approach the developer runs the program on the physical system. A benefit of this approach, which the previous two approaches do not have, is that the developer now has the product in a physical environment. As we discuss in the previous section, this will expose more bugs that are present in the system. As a result, the developer can make better estimates and observations about how their product behaves in a production setting.

A disadvantage of this approach, compared to approach 1 and 2, is that the developer is no longer working in a controlled environment, where everything is generated by a computer program and made available for analysis. Instead, observability on FPGA execution is low due to the limited number of IO on chips [36]. Another downside of this approach is that developers lose the ability to pause, step and break execution. And once a bug is observed, it can be tricky to pinpoint exactly what input caused the bug.

Observing a FPGA circuit can be done in multiple ways [21]. One way is via readback. Readback is a process that allows all registers within the FPGA to be read while the device is stopped [18] (section 2E). Readback is a useful feature, but a drawback is that not all FPGAs support it.

Another way is to use external logic analyzers (ELA), see figure 4.2. ELAs attach to pins on the FPGA in order to observe signals. The observability that comes from this is limited, as we only get to observe pin signals. ELAs do not observe the actual state of an FPGA, but the developer can temporarily route internal signals to external pins in order to increase ob-

Figure 4.2: An external logic analyzer

servability of the state. The functionality of an ELA can also be embedded into a circuit to remove the need for extra hardware. As an example, Alterna SignalTrap Megafunction [11] embeds a circuit into the design which connects output pins so that we can observe the state of the circuit.

A major drawback of approach 2 and 3 is that they operate on the hardware level. That is, the developer is observing hardware signals. This is not optimal, because the input for HLS compilers is a software language. So, the developer would need to link the observed hardware signals back to the source code. This is challenging because the circuit need not represent the source code in a trivial way. The optimizations made by the compiler add to the challenge. This problem was recognized and techniques have been developed to link hardware signals back to the software code, leading to approach 4.

4.4 Observing the circuit and linking observations back to the source code.

With the previous remarks in mind, we come to the realization that we need software level insight into real world execution. The techniques discussed in this section allow the system to run in hardware and therefore accurately capture the behavior of the system. This includes interaction with other components. Running the system at speed means encountering real world behavior and thus capturing real world bugs.

Readback (recall section 4.3) is not feasible here, because starting and stopping a circuit each time we want to capture the state will prevent the FPGA from interacting with other parts of the system. Instead debugging is done through instrumentation [19]. Instrumentation is the act of adding extra circuitry to provide visibility and/or controllability of the design. In practice, this means storing information on-chip inside a so-called trace buffer. A trace-buffer is a block of memory that stores information

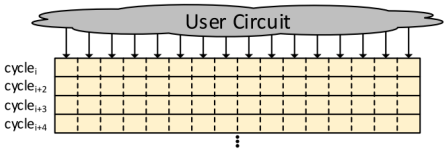


Figure 4.3: A trivial trace buffer that records all signals each cycle [18].

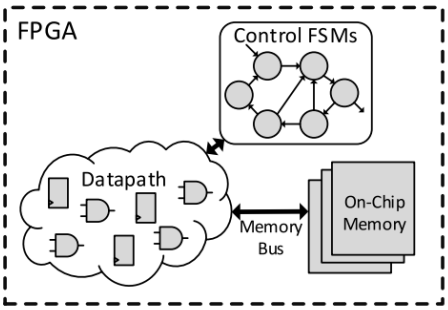


Figure 4.4: Typical HLS circuit structure [18].

about the state at certain times. This trace buffer can then be analyzed offline and linked back to the source code. Trace buffers are limited and circular, meaning that new values overwrite old values when the buffer is full.

The instrumentation of the circuit can be used in two ways [19], [18].

1. Interactive mode. The circuit is instrumented with hardware that allows for step and breaking, like software engineers are used to with GDB. When the circuit is stopped, all variables can be inspected by reading out the most recent values in the trace buffer. A disadvantage of this is that the circuit does not run at full speed and this might obstruct interaction with other system parts.

2. Replay Mode. The circuit is instrumented such that it runs at speed until a certain breakpoint is reached. After this, the (probably) full trace buffers are read out. Afterwards, the engineer can replay the execution and inspect variables. A disadvantage of this mode is that we lose full visibility because resources are limited.

Note that the trace buffers are read out to a regular computer, and interpreted by a computer program before shown to the engineer at a software level.

A trace buffer that captures all values at every cycle (see figure 4.3) is equivalent to an embedded logic analyzer. However, knowledge about the

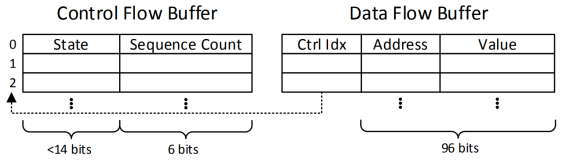


Figure 4.5: Split Trace Buffer Architecture [19].

circuit can be used to make the trace buffer store more information per Kb. Because on-chip memory is limited, this is of significant importance. A generic hardware ELA/trace buffer treats all signals the same. However, designs produced by an HLS tool tend to have a predetermined structure with easily identifiable important state bits and data bits. Often, C code is compiled into low level assembly-like code and then the HLS compiler maps control instructions such as jump to a finite state machine (FSM) and computation instructions such as 'add' to datapath logic [18]. This typical structure is visilized in figure 4.4. Highly optimized circuits are an exception to this pattern, but these are out of scope for a general purpose HLS debugger.

Because of this pattern, it is useful to have two buffers: a control trace buffer and a data trace buffer, and optimize them separately. This is called a split trace buffer architecture [19].

4.4.1 Control Trace Buffer optimization

The control trace buffer is used to store state bits of the FSM. The straightforward way is to store the state in each cycle. However, the state value often is only incremented by one. Especially when the program has a lot of sequential computations. Therefore a sequence count is stored next to each state, indicating the amount of sequential state updates. So if our state trace goes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 5, then state 1 is stored together with a sequence count of 9 and in the next row of the buffer we will have state 5. Programs that include a lot of control flow (functions, branches) will see less benefit of this optimization.

Another optimization is to leverage state information to "generate logic that captures only the necessary state bits" [19]. Sadly, the authors do not elaborate on this optimization any further.

4.4.2 Data Trace Buffer optimization

The data trace buffer is used to store signals that correspond to variables in the source code. Just like with the control flow, the straightforward way is to store every signal on each cycle. However, this is not feasible and instead signals are only stored when they change [19]. As a consequence, it is no

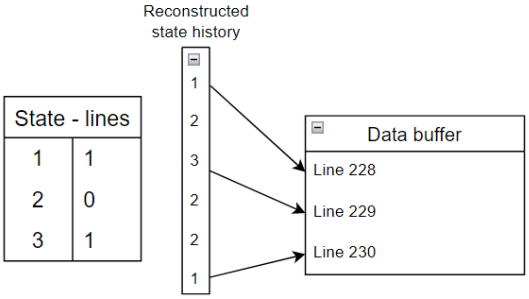


Figure 4.6: Reconstructing link between control and data without $ctrl_{idx}$.

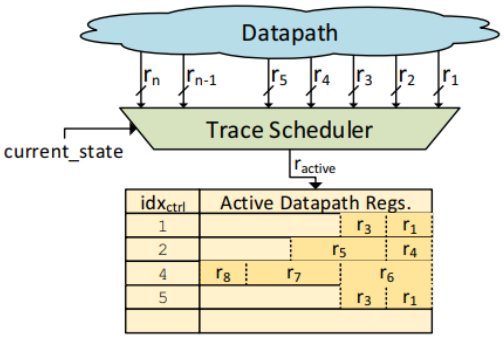


Figure 4.7: Trace scheduler [17].

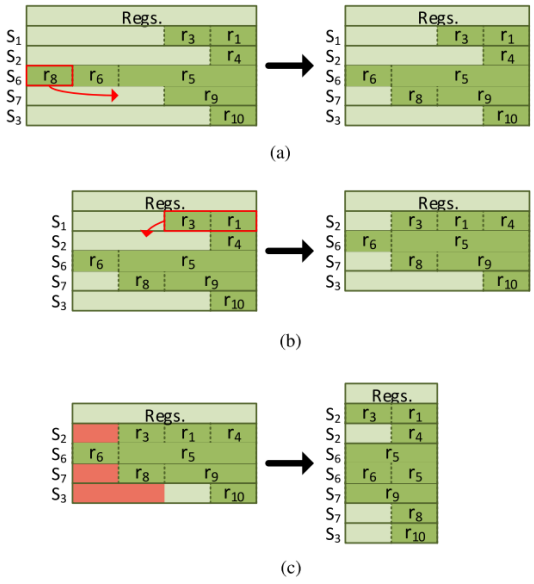


Figure 4.8: Three different trace buffer scheduling methods [18].

longer the case that there is exactly one cycle between each row in the data trace buffer. This is a problem as we need to combine the data and control buffers for analysis. To keep the link intact, an additional field called $ctrl_{idx}$ is added to keep track of when the change occurred. This is visualized in figure 4.5, as we can see the $ctrl_{idx}$ is used to link the data and control traces together.

However, the link between the control and trace buffer can also be restored without $ctrl_{idx}$ [18], this reduces buffer width as there is no more need for a $ctrl_{idx}$ field. Instead it is determined per state whether or not a line is added to the data buffer. The most recent state of the control buffer is the state the circuit was in when the traces were captured. By starting at this state and going through the data buffer in reverse, the link between control and data is restored.

Figure 4.6 depicts this process. The most recent state is state 1. We see that state 1 writes a single line to the data buffer. So line 228 belongs to the highest control buffer entry. We then see state 2, which writes zero lines. We go on to the next control buffer entry: state 3. State 3 writes one line and thus line 229 belongs to that control buffer entry. Then two times state 2 and finally a state 1 again, meaning line 230 must belong to the lowest control buffer entry. This algorithm is trivial to implement in code by iterating over the control buffer and keeping a pointer to the data buffer, incrementing it depending on how many lines are written by the current state.

Selecting signals to store in the trace buffer is done by circuitry called a Trace Scheduler (figure 4.7). Based on the state, only the relevant signals are combined into a single signal r_{active} and passed onto the buffer. As can be seen in this figure, quite some space is wasted. Therefore [17] suggests three different optimizations. Their advantages are best understood through a picture, see figure 4.8.

(a) *Delay-worst scheduling* As the name suggests, this optimization looks at the widest line. In the figure, this is S_6. To reduce buffer width, signals that are still available at a later time (r_8) can be delayed to a later state (S_7). In case a signal is not available anymore, for example when a signal is generated and used in the same cycle, an additional register may be added to the circuit. Determining what signals should be delayed is done by a greedy algorithm that repeatedly identifies the worst state and attempts to move the smallest signal to a later state. Note that the user circuit does not change and that the analyzer tool knows that the r_8 signal actually occurred at S_6.

(b) *Delay-all scheduling* Similar to delay-worst scheduling, but this attempts to move two states together into a single line, as long as it does not increase the width of the buffer.

(c) *Dual-ported scheduling* FPGAs typically support dual ported memory, which allows for two writes in a single cycle. By cutting the r_{active} in half, and storing both halves under each other, memory is saved whenever a state fills up less than half of the buffer's width. In the figure, only the entry for S_3 benefits from dual-ported scheduling. Note that the removal of $ctrl_{idx}$ is extra useful here, as we no longer need to store $ctrl_{idx}$ twice for dual ported memory writes. When dual ported memory is leveraged, the lines per state table in figure 4.6 could also contain 2 to indicate that the state uses dual memory to store signals inside the data trace buffer.

The final optimization we discuss is signal restoration [18]. Signal restoration moves online storage load to offline computation (of which there is no shortage). Consider $a = b + c$. Because a depends on b and c, the value of a can be reconstructed offline based on the recorded values of b and c. So, we do not need to record the value of a anymore. This optimization reduces the amount of data stored in the data buffer and it also reduces the logic needed in the Trace Scheduler. In a typical program, there are many such dependencies and we need an algorithm to choose which signals to trace. To do this, the problem is viewed as minimizing the amount of traced bits based on when the signals are available. This is then solved through integer linear programming; exact details about the constraints and goal function can be found in section V.B of [18].

	LoC per 100Kb	Area overhead	Time overhead	Features
ELA	275	0	0	Capture everything every cycle
[19]	1243	11%	Unknown	Split Trace Buffer Architecture Logging on change Sequence count
[17]	4322	10%	6.9%	Delay worst Delay all Dual ported
[18]	15369	10%	0%	Removal of $ctrl_{idx}$ Signal restoration

Table 4.1: Progress by paper. ELA stands for External Logic Analyzer. In this case it is the SignalTap II.

We have discussed the optimizations presented in [19], [17], [18] on a per topic order. It is also worthwhile to note the improvement per new paper, which we show in table 4.1. The results are obtained against the CHStone

benchmark [68]. While area and time overhead are well defined, the term lines of code (LoC) remains vague and no exact definition is given. This ambiguity is problematic because the CHStone benchmark contains relatively simple programs, such as aes and sha. However, when programming on a higher level, a function call to sha(input) should only count as one line as there is no use for the engineer to inspect the hundreds of lines of code inside that function call.

4.4.3 Streaming instead of storing

The techniques discussed in the previous section use on chip memory to store the signal trace. An advantage of this is that high observability can be obtained, because on chip bandwith is very high. However, on chip memory is limited and therefore trace length is limited too. An alternative solution is to stream the data to off chip memory [16]. Here, the new limitation becomes band with. The techniques that limit memory usage while capturing signals still apply.

4.5 Conclusion

Method	Conceptual level	Needs HW	Real World
1	Software level	No	No
2	Hardware level	No	No
3	Hardware level	Yes	Yes
4	Software level	Yes	Yes

Table 4.2: Overview of debugging methods.

In table 4.2 we give an overview of the methods discussed in this chapter. As we mention in the discussion of approach 3, approach 1 and 4 are the most practical for the HLS toolchain as they provide software level insight. Approach 1 can be of use when the developers do not have access to hardware and/or are in the early stages of development. However, we need the ability to observe real world behavior as production moves to deployment, leaving only approach 4 as sufficient. Approach 4 does suffer from limited trace lengths, forcing developers do go through multiple debug iterations. To improve debug efficiency, techniques stated in section 4.4 are leveraged to optimize the debugging circuitry (instrumentation) for memory usage.

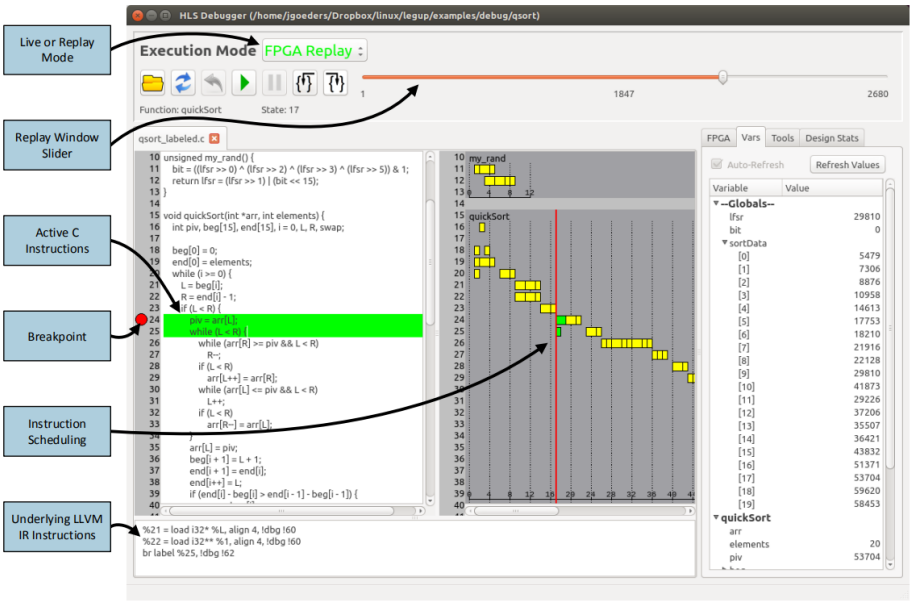


Figure 4.9: Software level hardware debugger [20].

4.6 Existing tools

The techniques discussed in approach 4 have been implemented in a proof of concept based on the academic HLS compiler LegUp [20]. It indeed supports two modes: a replay mode and a breakpoint mode. The goal is to provide a debugging interface similar to that of popular software debugging tools like GDB. An example of the interface provided can be seen in figure 4.9.

The debugging tool only supports hardware execution, and a heterogeneous HLS system debugger was introduced in [3]. This heterogeneous debugger uses the same hardware trace and has an additional software trace. Because of the sequential nature of software, any recording operation is likely to impact the performance of the system. Therefore the user must select, through source-code annotations, which parts of the software program should be recorded. The hardware and software trace are then combined to give the developer a single interface.

There are also other debugging tools such as Inspect [5], also implemented on LegUp, and a debugger made specifically for the now abandoned Sea Cucumber Compiler [24]. The HLS debuggers generally try to imitate software debuggers by implementing features such as single stepping, breakpointing, observing variables and setting variables. To do this, they utilize instrumentation. In case of a replay-window (e.g. breakpointing) there is a compromise between the amount of variables observed and for how long they are observed. In case of step-by-step features or setting variables, the

circuit cannot execute at full speed and this might cause some bugs to go unnoticed and/or limit interaction with other parts of the system.

Chapter 5

Verification, Testing and Performance Logging

Next to programming and debugging, there are other steps in development. This chapter discusses verification, testing and performance logging. Ensuring the functionality of a product as the responsibility of the engineer. However, the engineer should be able to view the compiler as a black box: it is the job of the compiler vendor to ensure correctness of the compiler. Especially in cases of a commercial compiler, where engineers do not even have access to the code of the compiler. In this chapter, we first discuss two techniques used to verify compiler correctness and then discuss verification, testing and performance logging in the context of HLS.

5.1 Compiler

Just as with software compilers, HLS compiler vendors employ various techniques to ensure correctness of their tool such as testing, formal verification, code reviewing and analysis. A technique that is unique to HLS compilers is cosimulation. In this technique, compilers test correctness of the generated circuitry based on the source code that is given by the developer. The process is depicted in figure 5.1. In this picture we take the C programming language combind with the GCC compiler, we also assume the HLS compiler outputs Verilog. These can be replaced by any matching software language/compiler and any HLD, respectively.

Initially, the C code is provided by the developer. Based on this, the HLS compiler generates Verilog code. But the C code itself can also be executed by compiling it to an executable. This gives us two functional units, one in software and one in hardware. By utilizing a Verilog simulator, both can be executed on the PC that the HLS compiler is running on. A computer generated test bench can be used as input to both and the two outputs can be compared. If they do not match then we can conclude that the HLS

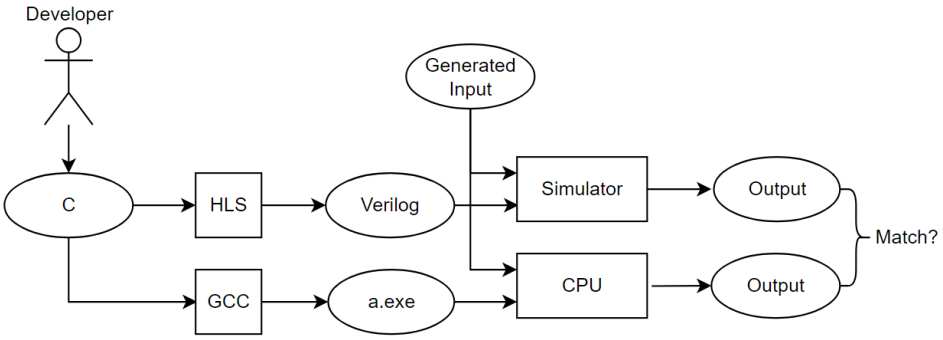


Figure 5.1: High level overview of cosimulation for HLS compilers.

compiler made an error in translating the C code to Verilog.

Cosimulation can also be used to test smaller components instead of the entire program, because the HLS compiler has knowledge about which Verilog code belongs to which part of the C code.

Cosimulation strictly tests the translating from the C code to Verilog. It does not test functionality of the system. As an example, consider the square function is implemented in C as follows:

```
unsigned long long square(signed x) {
    // Returns x^2.
    return 2;
}
```

It is clear that the code does not match the requirement of the function. However, cosimulation will not catch this as both the GCC and the Verilog will simply implement this function as returning 2 at all the time.

Cosimulation relies on correctness of the used software compiler. Recall from chapter 3 that HLS compilers use an already existing software compiler to compile the source code, and use the given result to generate HDL. If the software compiler is not correct, then the abstract syntax tree that the HLS compiler uses is incorrect in the same way, thus both the software and hardware implementation will have the same error, meaning their output will match despite being incorrect. However, this is not a bad thing as software compilers are already matured. Moreover, the alternative would be to have HLS compilers implement their own software compiler, which is a costly process and there is no guarantee that this implementation would be more correct than already existing ones.

Another technique to detect discrepencies between the software and hardware execution is to instrument the code to detect mismatches between software and hardware execution. AutoSLIDE [67] is a framework that does this for the LLVM compiler. When a mismatch occurs, AutoSLIDE inspects the datapath and pinpoints the corresponding lines of C/C++ code.

5.2 Software Code

As we mention in the previous section, it is the job of the HLS compiler to generate HLD based on a software language. Providing a correct software language input is the job of the developer. Proving correctness is done via testing and verification, and because the engineer is working with a software language, he/she can simply use already existing tools such as unit testing, doc testing and functional verification. HLS is therefore not different from software development when it comes to testing and verification.

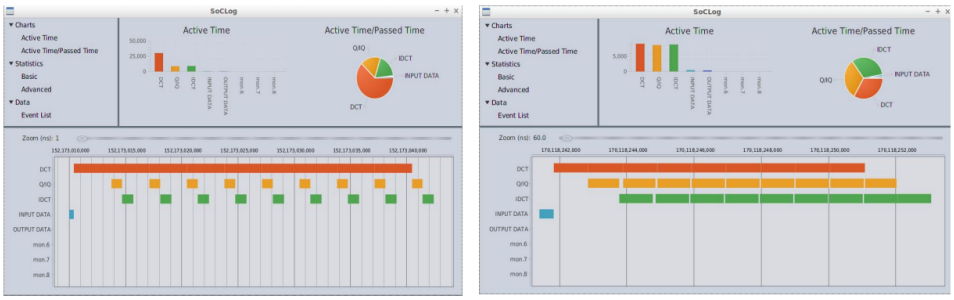
However, many HLS implementations have performance as their goal. To optimize for performance, a developer needs to be able to profile the execution in order to detect bottlenecks. To profile the FPGA execution we need insight into execution time of multiple modules that operate in parallel. In simulation it is easy to count the amount of cycles, but profiling is more accurate when the program is executed on actual hardware. A trivial approach to performance logging would be to print at the start and stop of functionality that needs to be profiled, similar to starting and stopping a timer. To reduce overhead, a developer can use a high performance logging framework such as HLS_PRINT [58], which is accurate in the range of microseconds and therefore also useful for profiling.

A more sophisticated approach is to automatically log activity triggers via instrumentation. The instrumentation is relatively simple, as profiling only cares about execution times and does not need to trace the execution as with the debugging techniques discussed in section 4. Activity triggers indicate when a module is working and when it is waiting. Based on this information, the developer can recognize which functionality is the bottleneck of the program and focus optimizing efforts accordingly. This instrumentation does come with some overhead, but we do not view this as a problem as the production version will not have performance logging instrumentation on it and thus run at maximum speed.

SoCLog [45] is such a HLS profiling tool. It has an activity log which can be read out at runtime in C/C++ source code via a function call to

```
int fpga_recv_log (fpga_t *fpga, int chnl,
                   void *data, long timeout)
```

SoCLoG comes with a GUI, see in figure 5.2. In this example there are three pipelined modules running on an FPGA. It is important to find the bottleneck (if there is one), because the bottleneck of a pipeline limits total throughput. In the figure on the left hand side one can easily identify the Discrete Cosine Transform (DCT) module to be the bottleneck. This observation incentivizes developers to optimize the DCT module. In this case the DCT module was optimized using loop optimizations of the Vivado HLS compiler. The optimized version is profiled and the result is shown on the



(a) (b)

Figure 5.2: GUI example of ScoLog HLS profiling tool [45]. Data flows through three components: DCT (red) → Q/IQ (orange) → IDCT (green). (a) Initially we see that DCT is dominating execution time, as a result Q/IQ and IDCT are underused. (b) After optimizations in the DCT component, the throughput for the overall system is enhanced.

right hand side. As we can see, all components run almost all of the time, thus throughput is no longer being limited by a single module.

HLScope [9] is another performance debugging tool used to identify bottlenecks in an HLS FPGA design. Their implementation does not have a GUI, instead the tool interactions with the user via the command line. The developer can see how many cycles a module takes and if this module is on the critical path of overall performance, which contains serially executing modules and the most time-consuming parallel modules. Based on this critical path, a stall rate is given by $\frac{\#\text{cycles for module}}{\#\text{cycles longest critical path}}$. Additionally, the reason for stalling is given. Possible reasons include memory access, waiting for data of another module (dependency stall) and waiting for a parallel-executing module to finish (synchronization stall). Based on this data the developer can target optimization efforts to specific modules in order to get a better performing system.

Chapter 6

HLS based applications

As the HLS toolchain is discussed in chapter 2-5, it is interesting to see what other researches have done with HLS. To survey this, we look at multiple HLS based applications on different domains. Although we cannot scan all papers related to a HLS based applications, this will give us a good idea of what HLS is used for in reality. Please note that we did not search for domains specifically, we only used general search terms such as "HLS" and "implementation". We first go over the applications by order of their domain and we end with some remarks.

6.1 Applications by domain

The first domain is audio signal decoding and encoding, which is done to reduce the amount of bits going over the network, see figure 6.1. The main objective in this domain is reducing latency. This way voice calls can be made with the smallest delay between talking and being heard. We consider encoder and decoders to be streaming applications, because they convert audio input into a binary or convert the binary into audio output during a live voice call. The used algorithms are non-trivial and therefore hard to implemented in HDL. The implementations we discuss all use linear predictive coding, which can benefit from the pipelining and parallelism of an FPGA.

The G.723.1 Decoder [34] and MELP Encoder [39] algorithms were implemented by the same authors using the Vivado HLS tool to convert C into Verilog and then Vivado's synthesizer to synthesize the Verilog onto the AMD Zynq-7000 ZC706 FPGA Evaluation board. The authors only reported the latency, with a maximum of 45k clock cycles for the G.723.1 Decoder and 1743k clock cycles for the MELP Encoder. The kit used has a 33MHz PS System Clock, so this comes down to a latency of 1.36 and 52.38 milliseconds, which are feasible delays for a live voice call. A CELP encoder [1] was implemented by other authors, who also chose the Vivado HLS compiler. All three implementations are evaluated to obtain area usage, but a

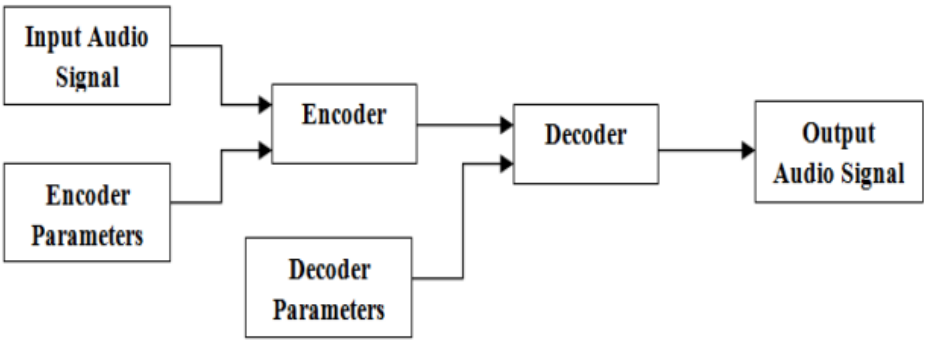


Figure 6.1: Audio encode/decode flow [34].

baseline is not provided, making it hard to draw any conclusions.

The domain of Neural Networks also benefits from FPGA strengths. Their accuracy comes at a high cost in terms of computational resources, and algorithms often include many nested for loops. Therefore the high performance and low power consumption of FPGAs are very useful. Because neural networks are an evolving research area, the configurability of FPGAs is convenient. The availability of HLS tools has enabled the development of complex neural networks algorithms on the FPGA. All neural network implementations we discuss were developed using the Vivado HLS tool and the C programming language.

Resnet50 CNN (Convolution Neural Network) is implemented on an FPGA [55]. The authors chose for an FPGA over the CPU because it lacks resources, over the GPU because it has a high power consumption and over ASICs because their development cycle is too long and expensive. The focus lies on loop optimizations, because the convolution operation consists of many inner for loops. Three optimizations are discussed (see also section III.D [44], mentioned in section 3.3.3).

1. Loop Tiling, dividing data into tiles that fit in the limited on-chip memory and processing tile by tile instead of all data in a single big loop.

2. Loop Transformation, reorder the loops so that data reuse factor is maximized and memory accesses are minimized. This way the HLS tool can apply optimization 3 better.

3. Loop Pipelining and Unrolling, allowing the next iteration to start before the previous one is finished and unrolling loops into independent operations which are then executed in parallel.

For evaluation the Resnet50 CNN implementation was ran on a Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit and achieved a speedup of

```
for (i = 0; i < K; ++i) {
  for (j = 0; j < K; ++j) {
    for (h = 0; h < N - K + 1; ++h) {
      for (w = 0; w < N - K + 1; ++w) {
      #pragma HLS PIPELINE
        for (f = 0; f < F; ++f) {
        #pragma HLS UNROLL
          for (c = 0; c < CH; ++c) {
          #pragma HLS UNROLL
            out[h][w][f] += W[i][j][c][f] * in[i+h][j+w][c];
}}}}}}}
```

Figure 6.2: Usage of loop optimizations in Vivado HLS code for convolution loop. [69].

198x compared to the Xilinx Zynq ZU7EV chip's processing chip running at 1.2 GHz.

Similarly, a signal modulation recognition CNN is implemented on the Xilinx XC7VX690T FPGA [69]. A code snippet can be seen in figure 6.2. As can be seen, the Vivado HLS compiler is instructed to use loop pipelining and unrolling. Compared to a GPU GTX1060, a 80.5% energy reduction and 28% speedup is reported. Another CNN implementation is discussed in [4], providing a more in-depth discussion on their system by going over the modules one by one. Just like the two previous applications, loop optimizations are heavily used. A Lenet-5 CNN for picture recognition [53] was implemented resulting in a 4.7x speedup and "much less power consumption" on a ZYBO Z7 FPGA compared to a 3.30GHz Intel Core i5x4590. Lastly YOLO CNN (using small bit integers) was accelerator using a Zynq 7z020 FPGA [26]. The implementation is compared to others works, see table 1 of [26], and it is shown that the HLS implementation outperforms implementations made directly in a HDL. The implementation has lower power usage (at least 16.7%) and higher performance (at least 4,86%) compared to others.

Next to CNN, we other machine learning applications also benefit from FPGA acceleration. Using Vivado HLS, Support Vector Machine calculation was accelerated [40], reporting a 10x latency improvement over other HLS implementations obtained by using a pipelined architecture that processes the input chunk by chunk, instead of storing all the data on the FPGA at once. The authors also report that their HLS implementation outperforms HDL implementations by a factor of 4.4. A Linear Discriminant Analysis classifier, an algorithm used to find a linear combination of features to separate the data into classes is implemented using HLS [48], [63] (for details

Figure 6.3: Image edge processing: original (left), Sobel (mid) [12] and Laplace (right) [54].

see section 2 in either of the references). FPGAs are efficient for this type of algorithm because of loop optimizations, array mapping and division optimization. Array mapping is an optimization where multiple small arrays are combined into a single larger one. Division optimization tries to reduce the amount of division operations at the cost of more multiplication operations, this is done by enlarging intermediate variables and replacing division by x by multiplication with $\frac{1}{x}$. Using these optimizations, [48] reports a 54.6% latency reduction while using less resources (details in their table II) using and [63] reports a 15% reduction in latency while achieving a 20% reduction in resource usage.

Image processing is another computation intensive domain which benefits from FPGA strengths. FPGAs are successful in image processing because they are able to exploit the inherently parallel nature of many image processing problems. A Laplace filter, an algorithm to sharpen the edges in images (see figure 6.3), is implemented using the HLS Tool AccelDSP [54]. The authors mention price, performance, power savings and ease of use as reasons to choose an FPGA as coprocessor. Their implementation ran on a Vertex-5 XC5VLX110T FPGA is 3.37x to 5.37x as fast as a 2.0GHz Intel Core 2 Duo CPU. The authors mention that this result is obtained without using any complex optimizations. However, this claim is a bit vague considering that the domain of AccelDSP is image processing. (The survey in [44] mentions that the now discontinued Accel project originated from MATCH, which has image processing as its domain, see table 1.) The same system is implemented using Verilog, resulting in similar results compared to the HLS implementations in terms of performance, but a roughly five times longer development time. Sobel edge detection, an algorithm to detect edges in images (see figure 6.3), is accelerated using Vivado HLS [37]. The hardware accelerated program performed 15.58x as fast as the Python only implementation which utilized OpenCV libraries.

Cryptographic calculations often rely on bit manipulation, which is why

ASICs and FPGAs outperform CPUs in this domain. Using Vivado HLS, the SHA-3 hashing algorithm is implemented in [32]. A typical SHA-3 round is computed through 24 rounds, where each round consists of five fairly simple operations: column parity, bitwise rotation, word permutation, bitwise row combination, and bitwise XOR operation with per-round constants. The aim for the SHA-3 implementation is maximum throughput, and this was initially 7 Mbps. By instructing Vivado HLS to use loop optimizations a throughput of 2000 Mbps was achieved. Quantum Schor's Algorithm is an integer factoring algorithm, known for its impact on the RSA algorithm, and is implemented in Vitis HLS [62]. The authors use a technique called matrix pruning for optimization and show their design is capable of factorising up to 8 digit ($\sim$27 bit) numbers. Which is rather disappointing as typical lengths for RSA keys are 512, 1024, 2048 and 4096 bits.

Lastly, various computation intensive tasks are accelerated through FPGAs, all using Vivado HLS. We only briefly mention them as their details focus on optimizing algorithms for which background knowledge is required. Related to cryptography, a Karatsuba modular polynomial multiplier is produced using HLS tools in [13]. In the field of computational physics we have an HLS optimizated implementation of the Tau Triggering Algorithm for data from the Large Hadron Collider at CERN [8] and an implementation of Chaotic Systems [59]. An HLS optimized version of Merge Sort is represented in [52] and an Adaptive Notch Filter implementation can be found in [7].

6.2 Remarks

Based off previous chapters, we expected the domain of HLS to be mainly computation intensive (streaming) applications and this turned out to be the case: the implementations we discuss all used the parallelism and pipelining that FPGAs are capable of. Together with the development in HLS tools the implementations showcase it is possible to implement abstract systems such as CNN and cryptographic algorithms with impressive performance in terms of throughput, latency and energy consumption.

The most used compiler is the Vivado HLS [66] compiler. This compiler is no longer being worked on, but of course developers can still use the latest version. The new version is called Vitis HLS [65]. It is interesting that none of the HLS based implementations in academic papers used an academic compiler, which confirms the notion that academic compilers are useful for compiler research but less useful in terms of actual development. However, there are many commercial compilers and none of the authors mention a reason for choosing Vivado or Vitis. This could be simply a matter of habit, but two advantages of Vivado and Vitis are that they can be used for free and that Xilinx provides a detailed tutorial which helps get developers used

to their HLS tool. Most FPGAs used were also manufactured by Xilinx, this is probably connected to the usage of the Vivado/Vitis HLS compiler (or the other way around). Again, none of the authors motivated their FPGA choice.

Lastly, the FPGA was mostly used in an accelerator context. This makes sense as data such as images and/or CNNs is not particularly useful on its own and systems using these techniques most likely also provide user interfaces and/or connect to other components, especially in the case of robotics.

Chapter 7

Future Work

The HLS toolchain has made significant advances in the past 20 years. However, many authors mention that is not as mature as the software toolchain. This comparison should be taken with a grain of salt, as the design space for HLS is significantly larger than the relatively simple assembly instructions of software. This makes it harder to develop robust tools that support lots of features while at the same time producing highly optimized circuitry. Nevertheless, aspiring the same capabilities is still a good target point, especially because HLS toolchain targets software engineers which are already used to these types of tools.

The FPGA hardware is fundamental to the HLS toolchain; without FPGAs there would be no need for HLS to begin with. However, in order to research FPGA architecture one would need to have a background in electrical engineering. As far as from the software side, the future work on FPGAs should aim to produce "better" FPGAs. That is, improve connectivity between CLBs, have more CLBs per FPGA and make CLBs more efficient in terms of power usage and the functionality that they provide. Of course these types of research is never finished, and manufacturers are expected to keep improving FPGA architectures.

Moving on to the programming side, it would be good to implement more features. As mentioned in chapter 3, most compilers take in the C languages which does not support OOP constructs. Functionalities such as virtual function calls do not inherently limit parallelism or pipelining, so might see OOP constructs added to the repertoire of HLS compilers. On the other hand, the HLS toolchain mainly targets computation intensive applications and these do not rely on OOP concepts to be implemented.

Many HLS compilers take an OpenMP approach to optimization, requiring the developer to manually insert pragma statements in the source code to get optimal performance. Future research is needed in order to have this done by the compiler itself. Even if this cannot be done completely automatically, it could be in the way of suggesting places for pragmas based on

static analysis or pattern recognition via machine learning.

The same goes for heterogeneous systems as discussed in section 2.3. For now, the user has to decide which part of the program is accelerated and which part is executed on the CPU. Future research is needed to automate this choice too.

Research into compilers can be done by taking one of the academic compilers and trying to extend them with a new feature. Next to that, the HLS toolchain will also benefit from optimizations in e.g. C compilers and steps in the synthesis process such as route and place.

Throughout the reading we have not encountered any mention of needed resources exceeding the available resources of the FPGA. Surely we cannot fit the entire Windows OS on a small FPGA chip, so there is a limit. Future work is needed to determine this limit. It is also interesting to research which software constructs are particularly costly in terms of area. Can one say anything about the cost of an if-statement?

The implementations discussed in chapter 6 are focused on strictly computational tasks with the only control flow being lots of for loops for summation across arrays. Similarly, the benchmarks such as CHStone [68] typically consist of small computational tasks. It is clear that an FPGA performs well in this case. However, general software is not represented by this. This raises the question if FPGAs perform well in programs with lots of if-branches and function calls. Future work is needed to determine if this is indeed the case, or if the FPGA can only function as an accelerator.

In regards to cryptographic HLS based applications, future research could try to execute/prevent side channel attacks on FPGAs. This is an ongoing research field and work has been done to perform attacks based on power usage [70], based on AI against an AES implementation [61] and to prevent timing attacks [42]. These attacks and defense measures are not specifically focused on HLS. Future research could be done to implement side channel attack prevention features for HLS compilers, essentially making a HLS compiler with cryptography as its domain.

We also observed that almost all implementation papers used the Vivado HLS compiler, this raises the question how commercial compilers relate to each other. A comparison between academic compilers has been done [44], but as far we know, there is no comparison between commercial compilers.

We encountered pretty sophisticated debugging tools that provide functionality close to GDB in chapter 4. However, the tools suffer from a limited replay window and/or limited observability. This makes it necessary for developers to do multiple for the developer to do multiple debug iterations. Future work is needed to improve visibility and window size in order to improve productivity.

The current HLS performance logging tools do not support heterogeneous systems. Future work is needed to combine the discussed techniques with already existing CPU profiling techniques in order to provide a single

profiling interface for the entire system.

Future work can also be done in implementing more algorithms with the HLS toolchain, as done in chapter 6. This will increase the knowledge around HLS systems and also delivers good results in terms of speedups and a lower energy consumption. To add to the diversity, it would be good to implement applications using a compiler other than Vivado or Vitis. Describing the use of other HLS tools (e.g. debugging, profiling) would also be an addition to the existing literature.

As FPGAs gain popularity, educational institutes that teach software engineering need to consider teaching courses in FPGA development through HLS. A mini-course for a hardware implementation of neural networks using HLS is documented and motivated [27]. More future work is needed to determine an efficient way of teaching HLS design.

Chapter 8

Conclusion

In this document, we have provided an introduction to and overview of the HLS toolchain. Fundemental to the the HLS toolchain are HLS compilers, most often taking in C/C++ and outputting Verilog/VHDL. In chapter 6, we saw that Xilinx' Vivado HLS and Vitis HLS were by far the most used compilers. This compiler has proven both functionality and usability and because of this, we recommend readers who wants to start developing FPGA applications using HLS to read the Vitis HLS tutorial [65].

We have seen that development steps that provide insight into execution, such as on-chip debugging and performance logging, require instrumentation of the circuit to gain the needed insight. However, steps that are about functionality, such as unit testing and functional verification, the already existing software tools can be used.

Because of the efficiency of FPGA, the need for high performance computing and the advances in HLS tools, we expect to see more widespread adoption of FPGA development via HLS tools in the coming years. This means that both companies and educational institutes will need to adapt to reap the benefits of FPGA development. We expect future work to focus on developing more features along the entire development process, with a focus on heterogeneous systems.

Bibliography

[1] Prakash C S Abhijna, N R Sangeetha, Jadav R Sagar, R Rahul, and Gaurav Gupta. Implementation of celp encoder using vivado hls. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 1443–1447, 2017.

[2] Inc Advanced Micro Devices. Ise simulator. https://www.xilinx.com/products/design-tools/isim.html, 2023. [Online; last accessed 25-7-2023].

[3] Matthew B Ashcraft and Jeffrey Goeders. Unified on-chip software and hardware debug for hls-accelerated programs. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 354–357, 2018.

[4] Darío Baptista, F. Morgado-Dias, and Leonel Sousa. A platform based on hls to implement a generic cnn on an fpga. In *2019 International Conference in Engineering Applications (ICEA)*, pages 1–7, 2019.

[5] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. Source-level debugging for fpga high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[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*, FPGA '11, page 33–36, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Tianbi Cao. Implementation and research of vivado hls's function on fpga. In *2022 3rd International Conference on Computer Science and Management Technology (ICCSMT)*, pages 240–243, 2022.

[8] Natalia Cherezova, Dmitri Mihhailov, Sergei Devadze, and Artur Jutman. Hls-based optimization of tau triggering algorithm for lhc: a case study. In *2022 18th Biennial Baltic Electronics Conference (BEC)*, pages 1–6, 2022.

[9] Young-Kyu Choi and Jason Cong. Hlscope: High-level performance debugging for fpga designs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–128, 2017.

[10] NATIONAL INSTRUMENTS CORP. Labview fpga compile worker compile time benchmarks. https://www.ni.com/en/support/documentation/supplemental/12/labview-fpga-compile-worker-compile-time-benchmarks.html, 2023. [Online; last accessed 25-7-2023].

[11] Altera Corporation. Signal tap embedded logic analyzer megafunction. https://flex.phys.tohoku.ac.jp/riron/vhdl/up1/altera/ds/signal.pdf, 2000. [Online; last accessed 25-7-2023].

[12] Ashish Fagna. Godbolt compiler explorer. https://medium.datadriveninvestor.com/understanding-edge-detection-sobel-operator-2aada303b900/, 2018. [Online; last accessed 6-8-2023].

[13] Michael J. Foster, Marcin Łukowiak, and Stanisław Radziszowski. Flexible hls-based implementation of the karatsuba multiplier targeting homomorphic encryption schemes. In *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pages 215–220, 2019.

[14] Copyright Free Software Foundation. Gdb: The gnu project debugger. https://www.sourceware.org/gdb/, 2023. [Online; last accessed 25-7-2023].

[15] Matt Godbolt. Godbolt compiler explorer. https://godbolt.org/, 2023. [Online; last accessed 6-8-2023].

[16] Jeffrey Goeders. Enabling long debug traces of hls circuits using bandwidth-limited off-chip storage devices. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–143, 2017.

[17] Jeffrey Goeders and Steve J.E. Wilton. Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, 2015.

[18] Jeffrey Goeders and Steven J. E. Wilton. Signal-tracing techniques for in-system fpga debugging of high-level synthesis circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):83–96, 2017.

[19] Jeffrey Goeders and Steven J.E. Wilton. Effective fpga debug for high-level synthesis generated circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[20] Jeffrey B. Goeders and Steven J. E. Wilton. Allowing software developers to debug HLS hardware. *CoRR*, abs/1508.06805, 2015.

[21] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging fpga circuits. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 41–50, 2001.

[22] Gunther, Milne, and Narasimhan. Assessing document relevance with run-time reconfigurable machines. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 10–17, 1996.

[23] J.R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 12–21, 1997.

[24] K.S. Hemmert, J.L. Tripp, B.L. Hutchings, and P.A. Jackson. Source level debugger for the sea cucumber synthesizing compiler. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 228–237, 2003.

[25] H.-C. Hsieh, K. Dong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, and R.H. Freeman. A 9000-gate user-programmable gate array. In *Proceedings of the IEEE 1988 Custom Integrated Circuits Conference*, pages 15.3/1–15.3/7, 1988.

[26] Jiaming Huang, Junyan Yang, Saisai Nui, Hang Yi, Wei Wang, and Hai-Bao Chen. A low-bit quantized and hls-based neural network fpga accelerator for object detection. In *2021 China Semiconductor Technology International Conference (CSTIC)*, pages 1–3, 2021.

[27] Nan-Sheng Huang, Jan-Matthias Braun, Jørgen Christian Larsen, and Poramate Manoonpong. Teaching hardware implementation of neural networks using high-level synthesis in less than four hours for engineering education of intelligent embedded computing. In *2019 20th International Carpathian Control Conference (ICCC)*, pages 1–7, 2019.

[28] Accellera Systems Initiative. Systemc standards & implementations. https://systemc.org/resources/standards/, 2023. [Online; last accessed 25-7-2023].

[29] Intel. Intel® high level synthesis compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html, 2023. [Online; last accessed 25-7-2023].

[30] Intel. Types of sycl fpga compilation. https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-0/types-of-sycl-fpga-compilation.html, 2023. [Online; last accessed 25-7-2023].

[31] Intel. Intel quartus prime software. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/article.html, Uknown. [Online; last accessed 25-7-2023].

[32] H S. Jacinto, Luka Daoud, and Nader Rafla. High level synthesis using vivado hls for optimizations of sha-3. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 563–566, 2017.

[33] Al-Shahna Jamal, Jeffrey Goeders, and Steven J.E Wilton. An fpga overlay architecture supporting rapid implementation of functional changes during on-chip debug. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 403–4037, 2018.

[34] M Koushik, Shashidhar Shivanagi, Gaurav Gupta, Jawed Qumar, and D. Saravanan. Implementation of g.723.1decoder on zynq fpga using hls. In *2017 International Conference on Inventive Computing and Informatics (ICICI)*, pages 263–266, 2017.

[35] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[36] R. Leatherman and N. Stollon. An embedding debugging architecture for socs. *IEEE Potentials*, 24(1):12–16, 2005.

[37] Han Sung Lee and Jae Wook Jeon. Accelerating image processing on fpgas using hls and pynq. In *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pages 1–2, 2020.

[38] E. Lemoine and D. Merceron. Run time reconfiguration of fpga for scanning genomic databases. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, 1995.

[39] Koushik M, Shashidhar Shivanagi, Jawed Qumar, Jyoti Yadav, and D. Saravanan. Implementation of melp encoder on zynq fpga using hls. In *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*, pages 87–91, 2017.

[40] Mohammad Amir Mansoori and Mario R. Casu. Hls-based dataflow hardware architecture for support vector machine in fpga. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 41–45, 2022.

[41] Mirsky and DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, 1996.

[42] Shyamapada Mukherjee, Swapnanil kr Saikia, Stuti Anand, Ritu Chouhan, and Hiresh Das. A counter measure to prevent timing-based side-channel attack on fpga. In *2021 6th International Conference on Communication and Electronics Systems (ICCES)*, pages 983–988, 2021.

[43] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, 2012.

[44] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[45] Ioannis Parnassos, Panagiotis Skrimponis, Georgios Zindros, and Nikolaos Bellas. Soclog: A real-time, automatically generated logging and profiling mechanism for fpga-based systems on chip. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[46] Meher Krishna Patel. Fpga designs with verilog. https://verilogguide.readthedocs.io/en/latest/, 2017. [Online; last accessed 25-7-2023].

[47] Meher Krishna Patel. Fpga designs with vhdl. https://vhdlguide.readthedocs.io/en/latest/, 2017. [Online; last accessed 25-7-2023].

[48] Dezhi Peng and Jin Sha. Efficient hls implementation of fast linear discriminant analysis classifier. *IEEE Embedded Systems Letters*, 13(4):214–217, 2021.

[49] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, 2013.

[50] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[51] QBayLogic. Clash. https://clash-lang.org/, 2023. [Online; last accessed 25-7-2023].

[52] P. Asha Rani, M C Chinnaiah, Apurva Kumari, G. Preethika, and Y. Prem Kumar Reddy. Hls based design and optimization of merge sort algorithm for high performance computing. In *2023 4th International Conference for Emerging Technology (INCET)*, pages 1–4, 2023.

[53] Dai Rongshi and Tang Yongming. Accelerator implementation of lenet-5 convolution neural network based on fpga with hls. In *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, pages 64–67, 2019.

[54] Mahendra Samarawickrama, Ranga Rodrigo, and Ajith Pasqual. Hls approach in designing fpga-based custom coprocessor for image preprocessing. In *2010 Fifth International Conference on Information and Automation for Sustainability*, pages 167–171, 2010.

[55] Muhammad Sarg, Ahmed H. Khalil, and Hassan Mostafa. Efficient hls implementation for convolutional neural networks accelerator on an soc. In *2021 International Conference on Microelectronics (ICM)*, pages 1–4, 2021.

[56] Siemens. Catapult high-level synthesis and verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/, 2023. [Online; last accessed 27-7-2023].

[57] Siemens. Modelsim hdl simulator. https://eda.sw.siemens.com/en-US/ic/modelsim/, 2023. [Online; last accessed 25-7-2023].

[58] Nupur Sumeet and Manoj Nambiar. Hls_print: High performance logging framework on fpga. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 390–390, 2021.

[59] Mobin Vaziri and Hadi Jahanirad. Highly efficient implementation of chaotic systems utilizing high-level synthesis tools. In *2022 30th International Conference on Electrical Engineering (ICEE)*, pages 501–506, 2022.

[60] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, 1996.

[61] Huanyu Wang and Elena Dubrova. Tandem deep learning side-channel attack against fpga implementation of aes. In *2020 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, pages 147–150, 2020.

[62] Juhi Wani, Tarun Kumar Allamsetty, Rushikesh Gherde, and Vanita Agarwal. Hls implementation of quantum shor's algorithm using matrix pruning. In *2022 Second International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, pages 1–4, 2022.

[63] Michael R. Wasef and Nader Rafla. Hls implementation of linear discriminant analysis classifier. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2020.

[64] Wikipedia. High-level synthesis. https://en.wikipedia.org/wiki/High-level_synthesis, 2023. [Online; last accessed 25-7-2023].

[65] Xilinx. Vitis hls. https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0090-vitis-hls-hub.html, 2023. [Online; last accessed 6-8-2023].

[66] Xilinx. Vivado 2020.1 - high-level synthesis (c based). https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html, 2023. [Online; last accessed 27-7-2023].

[67] Liwei Yang, Swathi Gurumani, Deming Chen, and Kyle Rupnow. Autoslide: Automatic source-level instrumentation and debugging for hls. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–130, 2016.

[68] Sinya Honda Yuko Hara, Hiroyuki Tomiyama and Hiroaki Takada. Chstone benchmark suite. https://www.jstage.jst.go.jp/article/ipsjjip/17/0/17_0_242/_pdf, 2009. [Online; last accessed 25-7-2023].

[69] Jian Zhao, Yaqin Zhao, Hongbo Li, Yun Zhang, and Longwen Wu. Hls-based fpga implementation of convolutional deep belief network for signal modulation recognition. In *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, pages 6985–6988, 2020.

[70] Yilin Zhao, Qidi Zhang, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama. Power side-channel analysis for different adders on fpga. In *2021 18th International SoC Design Conference (ISOCC)*, pages 367–368, 2021.