Synthesizing Behavioral Descriptions directly into Hardware Circuits

First Edition


# High-Level Synthesis Made Easy

By Benjamin Carrion Schaefer

LEARN HOW HIGH-LEVEL SYNTHESIS WORKS AND WHY IT IS IMPORTANT

# High-Level Synthesis Made Easy

**Synthesizing Behavioral Descriptions directly into Hardware Circuits**

April 26, 2023

highX Technologies LLC

High-Level Synthesis Made Easy



**Colophon**
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

**Publisher**
First printed in April 2023 by highX Technologies LLC

To all my teachers. The good, the bad and the ugly. I have learned from all of you.

# Preface

I started my career accelerating computationally intensive applications onto Field-Programmable Gate Arrays (FPGAs) using VHDL. These applications were mainly from the civil engineering domain to e.g., simulated large particle assemblies. These applications had embarrassingly large amounts of parallelism, thus, they were perfect candidates for hardware accelerations. Moreover, because these applications were initially developed using a high-level language like Fortran, Pascal or ANSI C, it made sense to look into methods that could directly synthesize these descriptions into the accelerated hardware. This is how I learned about High-Level Synthesis (HLS), first as a user and then moving to the other side as a developer of HLS tools. I first developed a basic HLS tool during my postdoctoral studies at UCLA called C2See. I basically developed the HLS tool that I would have needed to accelerated these civil engineering applications. I later joined NEC's Central Research Laboratory where I was a member of the CyberWorkBench team that developed a commercial HLS tool. This tool was initially developed as an in-house HLS tools for the NEC Electronics folks (semiconductor division) and is now commercially available to everyone.

This book is intended to teach anyone, either someone with deep hardware development skills, or someone with little prior hardware design knowledge alike to learn how to design and verify dedicated hardware modules using HLS. HLS converts untimed or partially timed behavioral descriptions into *efficient* hardware descriptions in either Verilog or VHDL that can in turn be further synthesized into gate netlists through logic synthesis. The word *efficient* is extremely important as nobody would build an integrated circuit (IC) using HLS if the generated circuit is larger, slower and consumes more power than using traditional methods based on low-level Hardware Description Languages (HDLs).

The book will start by describing what HLS is and its main benefits. It will then cover the theory behind HLS. Basically I will answer the question of how an untimed behavioral description that was originally intended to be compiled to machine code to be executed by a Central Processing Unit (CPU) is converted into a dedicated hardware unit. The book will cover different types of optimizations to allow the designer to generate the hardware circuit within the intended constraints, and finally it will cover different input languages and how they differ. In particular ANSI-C and SystemC.

I share this book in the hope that someone might find it useful. Happy reading.

*Benjamin Carrion Schaefer, Ph.D.*

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| ADC | Analog to Digital Converter |
| ADRS | Average Distance to Reference Set |
| AHB | Advanced High-performance Bus |
| ALM | Adaptive Logic Module |
| ALU | Arithmetic Logic Unit |
| AMBA | Advanced Microcontroller Bus Architecture |
| AMS | Analog Mixed Signal |
| ANN | Artificial Neural Network |
| ASAP | As Soon As Possible |
| ALAP | As Late As Possible |
| ASIC | Application Specific Integrated Circuit |
| AST | Abstract Syntax Tree |
| API | Application Programming Interface |
| BDL | Behavioral Description Language |
| BF | Brute Force |
| CDFG | Control Data Flow Graph |
| CGRA | Coarse-grain Reconfigurable Array |
| CGRRA | Coarse-grain Runtime Reconfigurable Array |
| CPU | Central Processing Unit |
| CWB | CyberWorkBench |
| DAC | Digital to Analog Converter |
| Data Initiation Interval | DII |
| DFG | Data Flow Graph |
| DMU | Data Manipulation Unit |
| DSE | Design Space Exploration |
| DSP | Digital Signal Processing |
| DVFS | Dynamic Voltage Frequency Scaling |
| DVS | Dynamic Voltage Scaling |
| EDA | Electronic Design Automation |
| FCNT | Functional Unit Constraint |
| FIR | Finite Impulse Response |
| FPGA | Field-Programmable Gate Array |
| FSM | Finite state machine |
| FU | Functional Unit |
| GA | Genetic Algorithm |
| GPU | Graphical Processing Unit |
| GUI | Graphical User Interface |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| IC | Integrated Circuit |
| IDE | Integrated Design Environment |
| IP | Intellectual Property |
| IR | Input/Intermediate Representation |
| LC | Logic Cell |
| LE | Logic Element |
| LUT | Lookup Table |
| ML | Machine Learning |
| PE | Processing Element |
| PI | Primary Input |
| PO | Primary Output |

| | |
|---|---|
| RTL | Register-Transfer Level |
| SA | Simulated Annealing |
| SDC | Sum of difference constraint |
| SoC | System on Chip |
| SPI | Serial Peripheral Interface |
| STC | State Transition Controller |
| STP | Stream-Transpose Processor |
| UART | Universal Asynchronous Receiver-Transmitter |
| VHDL | Very High-Speed Integrated Circuit Hardware Description Language |
| VLSI | Very Large-Scale Integration |

# Introduction 1

## 1.1 What is High-Level Synthesis (HLS)?



HLS can be defined as:

**The process of converting an untimed or partially timed behavioral description into efficient hardware.**

You can think of the process of converting ANSI-C, C++ or MATLAB into a synthesizable low-level Hardware Description Language (HDL) like Verilog or VHDL. I include in the definition *partially* timed behavioral description because as we will see later, many commercial HLS tools allow to time different portions of the behavioral description. This allows HLS user to e.g., model hardware interfaces.

Figure 1.1 shows an overview of the entire process. In particular, Figure 1.1 (a) shows the inputs to the process which are not only the behavioral description to be synthesized ($C_{in}$) but additional elements.

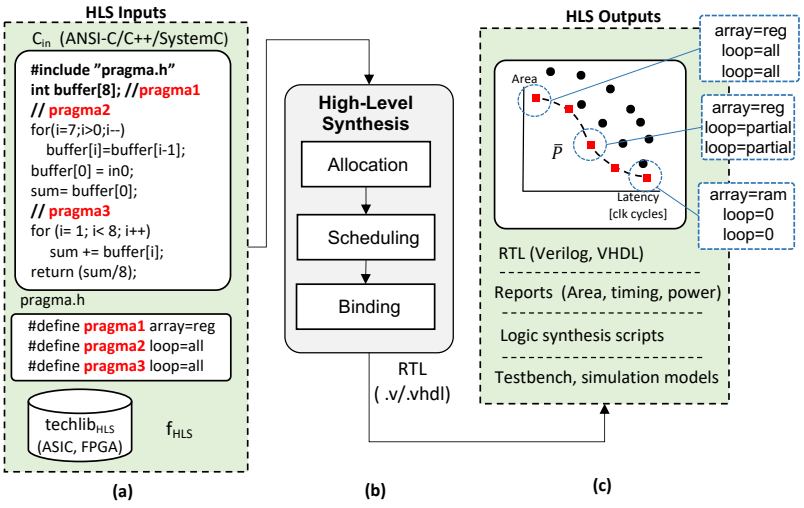



**Figure 1.1:** HLS overview. (a) HLS inputs; (b) HLS main steps; (c) HLS outputs.

In particular the inputs to any HLS process are the following:

**Behavioral Description ($C_{in}$)** This is the behavioral description to be converted into a hardware circuit through HLS. Different HLS vendors support different input languages. The most common ones are ANSI-C, C++, SystemC (C++ class for hardware design) or MATLAB.

**Synthesis Directives ($pragma_{HLS}$)** These are comments inserted in the behavioral description that allows the designer to control how to mainly synthesize arrays, loops and functions. These directives are extremely important in order to generate the hardware circuit with the desired constraints.

**Technology Library ($techlib_{HLS}$)** HLS requires a technology library that includes the area and delay of basic operators like adders, multipliers, multiplexers, etc. This library is extremely important to get good results. name.[1]

1: I will discuss in Chapter 3 how this library is generated.

**Target Synthesis Frequency ($f_{HLS}$)** The designer has to specify the target frequency at which the generated hardware needs to run as an input to the HLS process.

The HLS process reads in all of these inputs and performs three main steps as shown in Figure 1.1(b). In particular: (1) Resource allocation, (2) scheduling and (3) binding. These steps will be discussed in detail in Chapter 3.

The HLS process finally generates the hardware circuit specified in the selected HDL (Verilog or VHDL) as shown in Figure 1.1(c). In addition, the HLS process typically also generates additional outputs, In particular:

**Register Transfer Level Description (RTL)** This is the main output of the HLS process. The hardware circuit described using a HDL. In the figure every point in the trade-off curve represents a unique RTL description with different area vs. performance trade-offs. This is one of the advantages of HLS compared to using low-level HDLs. Changing e.g., the synthesis directives the HLS process will generate a different circuit.

**Quality of Results (QoR) Report** HLS tools also report the results in terms of estimated area, performance, critical path delay and often power. It should be noted that the values reported after HLS are only approximated values and should be interpreted with caution.

**Logic Synthesis Scripts** HLS is the first step in a series of additional steps required to build a complete hardware circuit. Thus, most HLS tools generate scripts to interface the HLS tool with the logic synthesis tool. These can be Application Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) logic synthesis tools, as the RTL code needs to be further synthesized into a gate netlist.

**Testbench** The generated RTL code although correct by construct needs to always be verified. Thus, most HLS tools also generated testbenches as an output.

Figure 1.1(c) also highlights one of the crucial advantages of HLS. It allows to generate hardware circuits with different area vs. performance trade-offs from the same behavioral description by simply setting different synthesis directives. Out of all the possible hardware circuits the ones marked in red squares (■) are the most important ones as they are what is called the *Pareto-optimal* (basically the smallest for a given performance). The process of finding these Pareto-optimal designs can be automated. This is often referred to HLS Design Space Exploration (DSE). Chapter 6 deals exclusively with this topic.

## 1.2 Why HLS?

One of the questions that you might ask yourself is, why do we need HLS considering that Verilog and VHDL seem to be very effective ways

to build hardware circuits? For this we need to review what is happening in modern computer architecture.

With the advent of Moore's law, most Integrated Circuits (ICs) are now heterogeneous System-on-Chip (SoCs). These SoCs contain a variety of different embedded processor, embedded memory, different peripherals (e.g., UART, SPI, I2C) and a multiple dedicated hardware accelerators (HWaccs), all interconnected through an on-chip bus.

Figure 1.2 shows an example of a heterogeneous SoC. The main reason why HLS has become part of most VLSI design flow is in particularly to design the hardware accelerators in the SoC. In this example the SoC is composed of different dedicated HW accelerators. The figure also shows that two of them are designed using HLS and one using traditional HDLs.

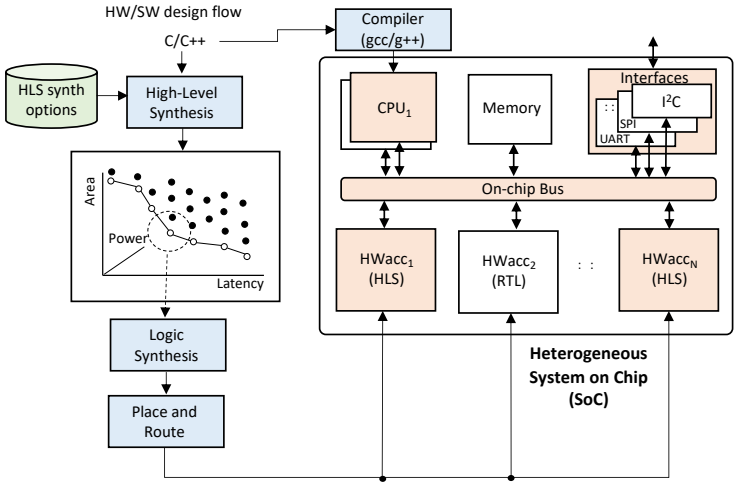



**Figure 1.2:** Example of heterogeneous SoC and its design flow.

One of the reasons why HLS is a good choice when designing these accelerators is that the applications to be accelerated in HW are often first developed using a high-level language such as ANSI-C, C++ or MATLAB. Some of these applications include encryption algorithms, Digital Signal Processing (DSP), image processing applications and more recently different types of Artificial Neural Networks (ANNs) to name a few.
The main characteristics of any application to be accelerated is that it should have large amounts of parallelism so that the dedicated hardware implementation can be faster and more energy efficient than executing it on the embedded SoC processor. It seems therefore natural to use HLS to directly synthesize any of these applications into a dedicated HW accelerator using HLS instead of having to manually translate the algorithm from the original behavioral description into RTL.

Figure 1.3 shows the performance per Watt advantage of these dedicated HW accelerators compared to general purpose Central Processing Units (CPUs) and Graphics Processing Units (GPUs).

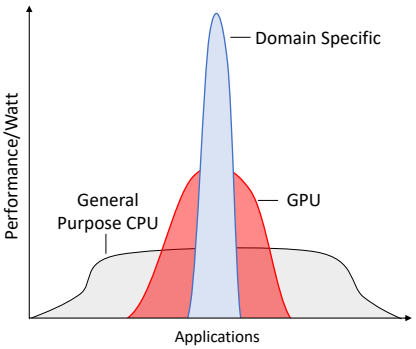



**Figure 1.3:** Advantages of domain specific computing.

CPUs can execute a large variety of applications efficiently, but have a poor performance per Watt response. GPUs can execute efficiently a much narrower family of applications, e.g., image processing and AI applications, but have a better performance per Watt response as they can parallelize well these types of applications. Finally, the dedicated

HW accelerators are fixed HW modules and can only execute a single application, but have the advantage of a very good performance per Watt response. This is one of the main reason that modern SoCs contain an increasing number of dedicated HW accelerators.

One of the problems building these heterogeneous SoCs is that the design and verification complexity increases significantly. Some SoCs like Apple's A14 SoC has over 11 billion transistors and include a larger number of dedicated hardware accelerators. Designing these SoCs using low-level HDLs is slow and error prone. Thus, new design methodologies are needed.

This is not new though. Figure 1.4 shows how the VLSI design methodologies have changed over time in the quest to increase the design productivity.

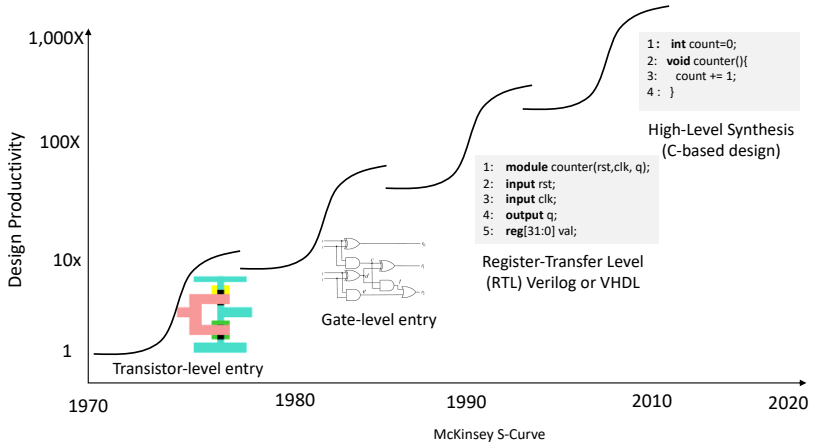



**Figure 1.4:** Progress of VLSI design methodologies over time.

Initially ICs were developed by manually *drawing* the transistors. Then gates were connected together in a scratchpad to build the circuits until the 90's where Hardware Description Languages (HDLs) like Verilog and VHDL were created. These HDLS relied on novel logic synthesis tools to convert these HDLs into efficient gate netlist. Although still being widely used, since the mid 2000's HLS has been steadily being adopted by most companies to design different portions of the ICs, mostly the HW accelerators.

One question that I will address in the next section is why HLS increases the VLSI design productivity.

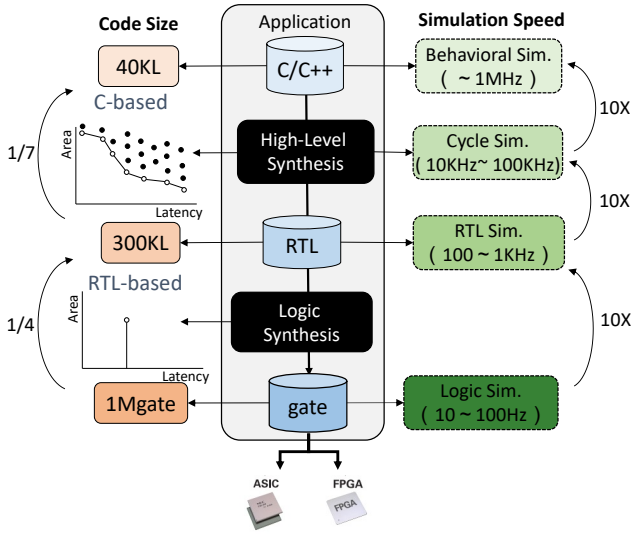



**Figure 1.5:** Summary of advantages of HLS compared to RTL and gate netlist.

## 1.3 Benefits of HLS

Raising the level of HW design abstraction from the RT-level (Verilog or VHDL) to the behavioral level has multiple advantages that I will highlight in this subsection.

Figure 1.5 shows an overview of some of the advantages that we quantified previously [1]. In particular:

**Advantage 1: Faster and Easier Design:** Writing C/C++ code is much easier than writing Verilog or VHDL. It requires less lines of code as C/C++ does not require to specify many things that are needed in any HDL. Some include the clock and the reset.

Figure 1.6 shows an example of a 32-bit counter specified in Verilog (RTL) and in ANSI-C. RTL code requires many more lines of code than the ANSI-C description, although both descriptions end up generating the exact same counter.

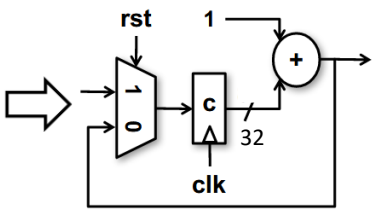
```
1  module counter(rst,clk, count);
2  input rst;
3  input clk;
4  output count;
5  reg[31:0] val;
6
7  always @ posedge clk)
8   begin
9   if (rst == 1b'1) begin
10     val<=32'h00000000;
11  end
12  else begin
13     val <= val+1;
14   end
15
16  assign count = val;
17  endmodule
```

RTL - Verilog

```
1  int count=0;
2  void counter(){
3    count += 1;
4  }
```

ANSI-C

High-Level Synthesis




32-bit counter

**Figure 1.6:** Source code differences of 32-bit counter in Verilog and ANSI-C.

One question that you might ask yourself now is, how does HLS generate the clock and reset and how do we specify if we want to have an

asynchronous or synchronous reset or a rising or falling edge clock. For this commercial HLS tools use synthesis options that need to be set when HLS in invoked. Unfortunately, these options are all tool dependent and every vendor has a different way to specify these options. As shown in Figure 1.5, we measured that 1 line of C code generates on average generated 7× the number of gates that a single line of Verilog or VHDL code generates. This represents a 7× increase in design productivity as writing less lines implies finishing the design and verification process faster.

**Advantage 2: Faster Verification:** Starting at the untimed behavioral level also allows us to speed up the verification process as different types of simulation models at the different abstraction levels can be leveraged. E.g., the untimed C/C++ description can be compiled using a regular SW compiler like gcc or g++ leading to simulation speeds of 1MHz. HLS also allows to generate fast cycle-accurate simulation model. These are again 10× faster than an RTL simulation and are timing accurate.

**Advantage 3: Ability to generate Functional Equivalent Designs:** One additional and unique advantage of HLS is that it allows to generated a variety of functional equivalent hardware implementations from the same behavioral description by simply setting different HLS synthesis options. I mentioned previously that one of the inputs to the HLS process are the synthesis directives in the form of pragmas. Setting different combinations of these pragmas lead to designs with different area vs. performance and power trade-offs. Using low-level HDLs for HW designs implies that as shown in Figure 1.5 the micro-architecture of the HW is fixed. This implies that the area and performance are fixed. Generating a new architecture with different trade-offs would involve having to fully design and verify the RTL. This is time-consuming and error prone.

## 1.4 HLS tools

There many commercial HLS tools available, and even some open-source ones. Chapter 9 reviews these in detail. Although these tools all convert behavioral descriptions into RTL it is important to understand the differences between these tools.

One important distinction is some target ASICs while others target FPGAs. Although ASIC HLS tools can also target FPGAs, FPGA vendors further optimize their HLS tools to make better use of the heterogeneous FPGA resources (e.g., BlockRAM and DSP macros). Furthermore, these tools are tightly integrated with the rest of the FPGA tool flow. One salient advantage of FPGA HLS tools is that they are available for free while ASIC HLS tools are normally very expensive.

It is also important to understand that each HLS tool has a different default synthesis mode. Some HLS vendors try to parallelize the behavioral description as much as possible by default, while others do not do any optimizations by default.
The quality (size and delay) of the generated circuit is also very different between commercial tools. This is why most companies adopting HLS as part of the VLSI design flow, especially the ASIC companies, perform

exhaustive evaluations between the different commercial tools before they commit to one.

## 1.5 Conclusions

In this chapter I have introduced the concept of HLS and why it has become more relevant now. I have also listed some of the main benefits of raising the level of VLSI design abstraction from the RT-level to the behavioral level. The next sections will describe in detail how the HLS process works.

**Summary**

- High-Level Synthesis is mainly used to design the hardware accelerators in modern heterogeneous SoCs.
- HLS increases the design productivity as it implies writing less lines of code.
- The verification process is faster with HLS due to the fast simulation models that HLS can generate.
- There are many commercial HLS tools available. It is important to understand their strengths and weaknesses.

# Hardware Platforms: From ASIC to FPGA

# 2



## 2.1 Introduction

There are two main options when building dedicated hardware circuits. The first is to build the complete chip from scratch, building an Application Specific Integrated Circuit (ASIC). These ASICs have the highest possible performance and consume the least amount of power as they are completely tailored to the application domain.

The other option is to use a field-programmable chip that has already been fabricated and then configure it with the specific hardware function that it should perform. There are different types of these field-programmable chips, but the most common ones are island style fine-grained Field-programmable Gate Arrays (FPGAs). Field-programmability refers to the ability of these chips to be reprogrammed to perform different logic functions.

Figure 2.1 shows these chips sorted by their complexity.
The simplest field-programmable devices (FPD) are Programmable Logic Array (PLA) and Programmable Array Logic (PALs). These are basic devices that allow to generate and logic function. The PAL consists of a programmable AND gate array and fixed OR array of vice versa, while the PLA has both arrays programmable.

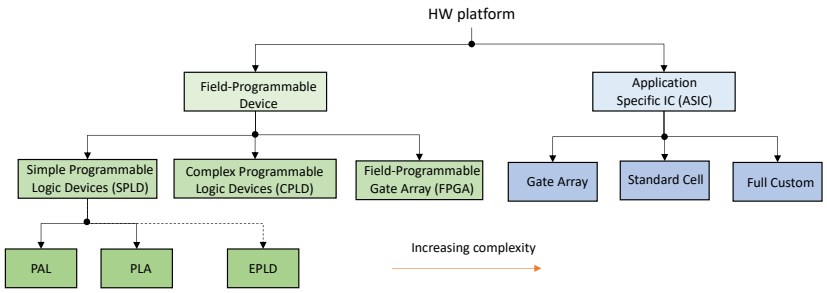



**Figure 2.1:** Classification of Field-programmable and Application Specific ICs.

On the other side of the spectrum, the most complex chips are full-custom ASICs where the transistors are manually created to maximize the performance and power savings.

## 2.2 Application Specific Integrated Circuits (ASICs)

ASICs are custom integrated circuits, designed for a specific task. As shown at the classification Figure 2.1 there are different types of ASICs.

**Gate Array** are basically fixed ASICs that are built from a regular gate array structure like an FPGA, but where the reconfigurability of the array is removed to reduce the area and delay. As we will see in this chapter, the logic that makes FPGAs reconfigurable are the main

reason for the large area and performance overheads of FPGAs. Removing this logic makes the IC faster, while being able to benefit from many of the benefits of FPGAs like being easy to program.

**Standard Cell** is the most common ASIC. The design starts with a HDL and logic synthesis tools that synthesize this HDL description with a given pre-characterized technology library into an optimized gate netlist. This gate netlist is then placed and routed and GDSII masks generated to fabricate the circuit. The technology library contains the detailed information of the basic gates that are supported like AND2 (two-input AND gates) or AOI (AND-OR-INVERT). This information includes the area, delay and power of this basic gates.

**Full Custom** design implies, every single gate is manually created and optimized. This is obviously extremely time-consuming and error prone, but leads to the most optimized of all circuits. Full custom design only makes sense for ICs that will be sold in extremely high volumes, e.g., micro-processors.

## 2.3 Field-Programmable Gate Arrays (FPGAs)

Field-programmable devices (FPD) were first introduced in the 70's. Their main purpose was to *absorb* all the glue logic that electronic systems had dispersed around their systems into a single chip.

Figure 2.2 shows the trend of how these FPD have evolved over time. Initially they could only execute combinational logic functions, through a programmable array or AND or OR matrix.

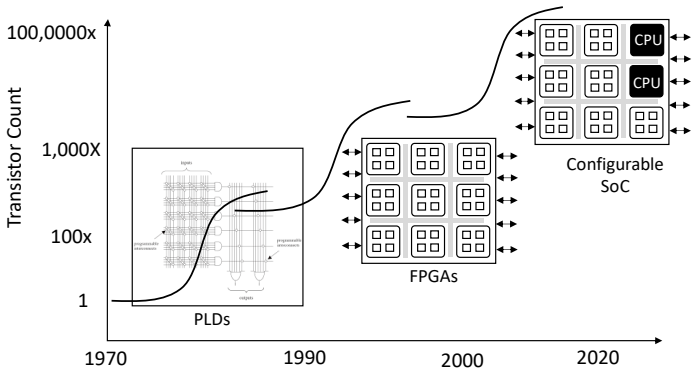



**Figure 2.2:** Trend of. Field-programmable devices over time.

Look-up table-based Field-Programmable Gate Arrays (FPGAs) where first introduced by Xilinx in 1984. The architecture completely changed compared to previous PAL structures generating current island-style FPGAs where LUTs are surrounded by configurable interconnects and programmable IOs. CMOS technology scaling now made it possible to implement larger systems on these FPGAs expanded significant their application based from glue logic to complete hardware subsystems.

In 1983 Altera corporation was founded in California, following the footsteps of Xilinx. Altera came from the word "Alterable" recognizing the programmability of their chips. Since their recent acquisitions, Altera by Intel and Xilinx by AMD, they both consistently pushed the FPGA market.

Moore's law obviously also applies to FPGA and newer FPGAs now have billions of transistors and incorporate embedded processors and multiple different peripherals. These newer FPGA have been coined as Configurable Systems-on-Chip (CSoCs).

## 2.4 ASIC vs. FPGA

It is important to understand the economic differences between ASICs and FPGAs. ASICs require a team of very skilled specialized hardware design and verification engineers. These devices are so complex that a single person cannot develop the entire design. Expensive Electronics Design Automation (EDA) tools are also needed. These are extremely complex tools that can synthesize HDLs into gate netlists, place and route these gates and generate the fabrication masks for the ASIC. Also complex verification tools are needed. Because ASICs need to be fabricated in expensive fabs and re-spins are very expensive, the design and verification teams need to ensure that the chip will work on the first try (as much as possible). For this, often expensive emulation equipment is needed. This makes the non-recurring engineering (NRE) costs of ASICs very high.

On the other hand, FPGAs have been already fabricated and tested and are available off-the-shelf. FPGA tools vendors also provide the tools to design and verify the FPGAs freely available. Prototyping FPGA boards are also inexpensive making it very easy to fully prototype the hardware system. But then, why would anyone want to build ASICs instead of using FPGAs? There are two main reasons:

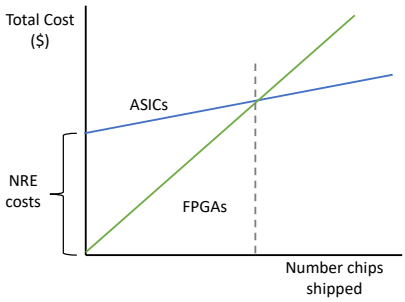



**Figure 2.3:** Costs of ASICs vs. FPGAs.

**Performance and Power:** The first is performance and power wise. ASICs are typically 10-100× faster and lower power then FPGAs. If the application requires extreme high performance and/or low power, then FPGAs might be good enough. The ability to reconfigure the hardware comes at a cost. The reconfigurable routing fabric slows down the FPGA significantly consuming also a lot of power.

**Economical:** FPGAs have very little NRE costs, but the unit cost is much higher than that of an ASIC. For high-volume product as shown in Figure 2.3 there is a break-even point where it starts making more sense to build ASICs rather than using FPGAs. This obviously considers that FPGAs meet the technical specifications required by the hardware systems, and that both alternatives are acceptable.

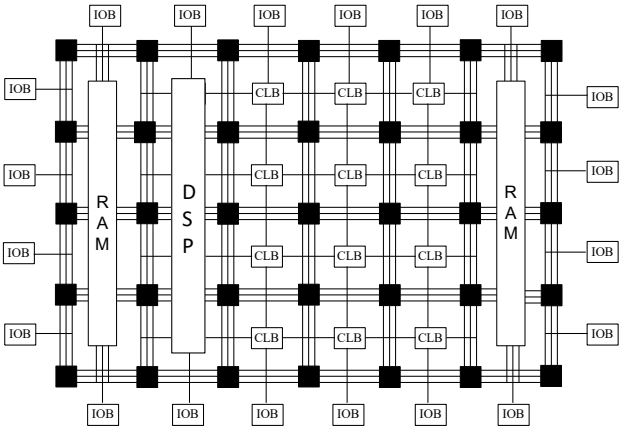



**Figure 2.4:** FPGA structure composed of CLBs, configurable routing, IO blocks, BlockRAM and DSP macros.

## 2.5 FPGA structure

Figure 2.4 shows an overview of traditional FPGAs. These island-based FPGAs are mainly built of a matrix of LUTs onto which to the logic function to be executed is mapped, all interconnected through the configurable routing resources. The LUTs are often grouped together into larger blocks called Configurable Logic Blocks (CLBs) - Xilinx notation. I should note that Intel and AMD/Xilinx FPGA structures are very similar, but that their naming convention is often different. This array of CLBs is surrounded by configurable IO blocks that can be configured as inputs, outputs or bi-directional IOs.

Every FPGA now also includes additional resources like embedded RAM (BlockRAM - Xilinx notation) and DSP macros. DSP macros were initially embedded multipliers of specific bit widths (e.g., 18/24-bit multipliers), and in newer FPGAs are multiply-accumulate (MACs) units. The main reason for having embedded multipliers in FPGAs is that FPGAs are often using in DSP applications that require many multiplications. These multipliers would consume too many CLBs if mapped onto the reconfigurable fabric, which would also slow the circuit down, as most of the delay in FPGAs is due to the reconfigurable fabric.

The next subsections will cover the different parts of the FPGA in detail.

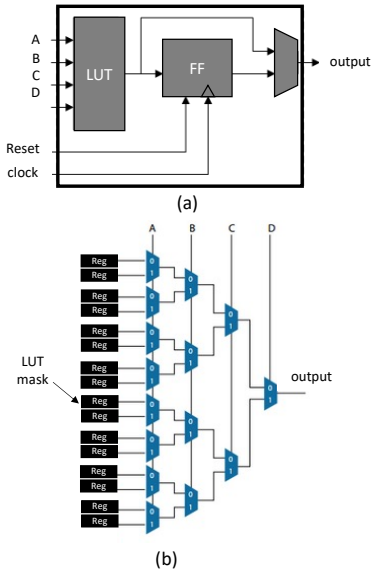



**Figure 2.5:** (a) Structure of Logic Element/Cell composed of 4-input/1-output LUT and FF. (b) Implementation of LUT using muxes.

### 2.5.1 Logic Cell/Element

The main component of any FPGA is the reconfigurable fabric. At the heart of the fabric is the logic element (LE) or logic cell (LC) onto which a logic function is mapped. This LE contain a 4-input/1-output LUT and a flip-flop that can be used or bypassed.

Figure 2.5(a) shows an overview of these LE, with Figure 2.5(b) showing how the 4-input LUT is typically implemented. Basically, any 4-input, 1-output logic function can be mapped onto a single LUT, where the LUT mask is fixed based on the truth table of the logic function and the control signals of the muxes acting as selectors for the truth table entry.

The LUT mask is generated during the synthesis process which generates the bitstream to includes this mask among other things that we will review later in this chapter

### 2.5.2 BlockRAM

Block RAM or embedded RAM are dedicate RAM blocks within the FPGA. As shown in Figure 2.4 these dedicated RAM blocks are inserted as columns within the reconfigurable resources.

Figure 2.6 shows the structure of a 16Kbit Block RAM from a Xilinx Virtex 4 device. Each Block RAM column in the FPGA contains multiple of these blocks that can in turn be configured as a single large memory with different organizations (width and depth). The memory can also be configured to act as synchronous or asynchronous memory, operate as a single port memory or a dual-port memory. As shown in in the figure the block RAM shown has two ports, Port A and Port B. This allows to read and write to the memory at the same time and hence, minimizing the memory bottle neck that most hardware systems have.

The amount of embedded memory available on an FPGA depends on the FPGA family and the device within the family. It is therefore important to know what kind of applications will be mapped on the target FPGA to evaluate the embedded memory needs.

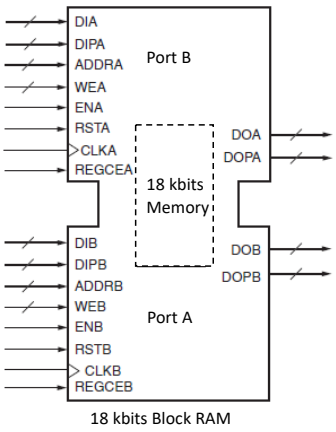



**Figure 2.6:** FPGA embedded RAM (Xilinx).

### 2.5.3 DSP macros/blocks

DSP macros (Xilinx notation) or DSP blocks (Altera's notation) used to be dedicated multipliers inserted similarly to the embedded RAM as columns within the reconfigurable fabric. The first FPGAs with DSP macros had 18×18-bit embedded multipliers distributed across the fabric.

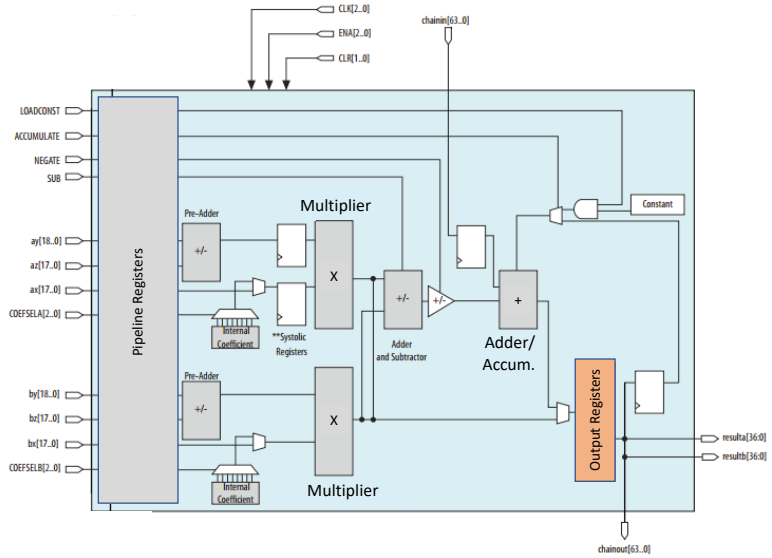



**Figure 2.7:** Block diagram of adaptive DSP block at Intel Stratix 10 FPGA. (Intel Stratix 10 Variable Precision DSP Blocks User Guide).

The main reason for having dedicated multipliers is that many applications amenable to be efficiently mapped to an FPGA required many multiplications. Some applications include DSP, video compression etc. Performing multiple multiplications in parallel significantly speeds up these applications. The main problem with first generation FPGAs without these dedicated multipliers was that multipliers require many LUTs, occupying most of the FPGA reconfigurable resources by themselves. Moreover, multipliers mapped onto the reconfigurable resources made them very slow.

Newer FPGAs' DSP macros are much more complex these days. They contain variable precision modules that have been optimized to efficiently compute different types of high-level constructs, e.g., FIR filter taps. They now include multiple multipliers, an output accumulation stage, different pre-computation stages, internal coefficient storage. These DSP macros can also be configured to map floating-point multipliers on them efficiently.

Figure 2.7 shows a block diagram of one of these adaptive DSP blocks from Intel's Stratix 10 FPGA

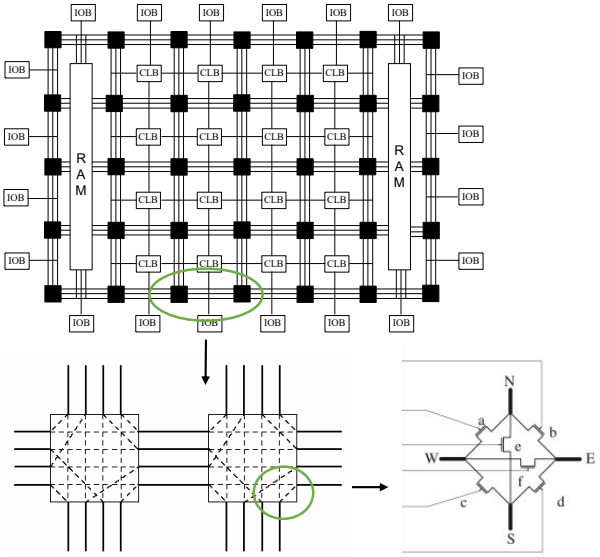



**Figure 2.8:** FPGA routing structure through configurable switch boxes.

### 2.5.4 Reconfigurable Interconnect

One of the main characteristics of FPGAs is that any logic function can be mapped on them. The configurable LUT just seen is one enabler. The other is the reconfigurable interconnect. This enables the connection between multiple LUTs, allowing the creation of larger circuits.

Figure 2.8 shows an overview of the main structure of the reconfigurable interconnect. It is based on switch boxes that have pass transistors. Enabling or disabling these pass transistors opens or closes the wire connecting any of the routing tracks.

The place and routing stage in the synthesis process will determine which LUTs will be used and which interconnect tracks to be used. The final bitstream programs the FPGA by enabling the different pass transistors.

One problem with this architecture though is the delay introduced by these switch boxes. This delay is extremely larger mainly due to the high parasitics (resistance, capacitance and inductance), dominating in many cases the overall circuit delay, and hence, the maximum frequency at which the circuit can operate.

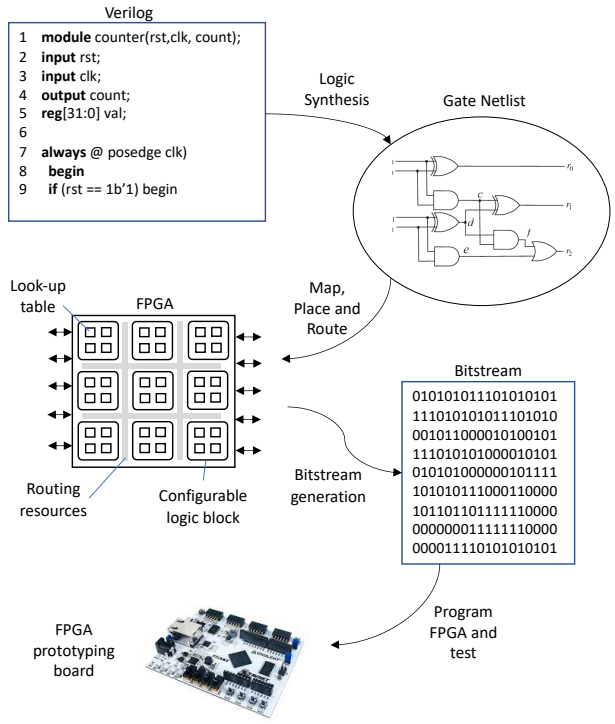



**Figure 2.9:** FPGA synthesis flow overview.

### 2.5.5 FPGA synthesis Flow

Figure 2.9 shows an overview of the complete FPGA synthesis flow starting from a Verilog description that needs to be mapped onto the FPGA. The process starts by synthesizing the Verilog code into a gate netlist through logic synthesis. This gate netlist is then mapped to the FPGA resources through a process called technology mapping and then placing and routing those resources onto the FPGA. Finally the bitstream to program the FPGA is generated.

Here is summary of all the steps in detail:

**Logic Synthesis:** Converts the RTL code (Verilog or VHDL) into an efficient gate netlist.

**Technology mapping:** Maps the gate netlist onto FPGA resources. This includes the IOs, the LUTs, DSP macros and BRAM.

**Place and Route:** This step determines which LE of the FPGA to use, based on different constraints like minimizing the wire-length between connecting to minimize the maximum delay.

**Bitstream Generation:** This final step generates the FPGA configuration file that is used to program the FPGA.

Figure 2.10 shows an example of AMD/Xilinx's synthesis flow pane, indicating if the different stages had been performed successfully. Observing this figure we can observe that an additional step is specified: Translation. This step merges all the input netlists and converts them to AMD/Xilinx own native generic database file format (NGD).

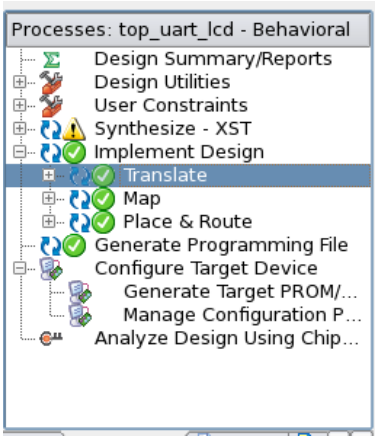



**Figure 2.10:** Xilinx ISE synthesis flow example

### 2.5.6 Logic Synthesis

Logic synthesis can be defined as:

**The process of converting a hardware description specified at the Register Transfer level (RTL), e.g., Verilog or VHDL into an optimized gate netlist given certain constraints and a target technology library.**

The constraints given in logic synthesis are typically timing constraints like maximum critical path delay or logic constraints like maximum fanout and maximum logic levels.

The technology library contains all of the gate level primitives characterized by their area, timing and power. In the ASIC case, these technology libraries are given in liberty (.lib) or data base (.db) formats. The main difference between these two library formats is that the liberty format is text form, and thus, readable and editable, while the db format is given in binary mode, making it more compact but cannot be read or edited.

Figure 2.11 shows a simple logic synthesis example where a multiplexer is described in VHDL. Next to the VHDL code we can observe the inferred hardware structure that the user wants to generate. Below is the gate netlists generated by the logic synthesis process. An initial gate netlist is unoptimized. This first translation step converts the RTL code into a gate netlist with primitives available in the technology library. This netlist is then further optimized to meet with the given timing and logic constraints as shown.

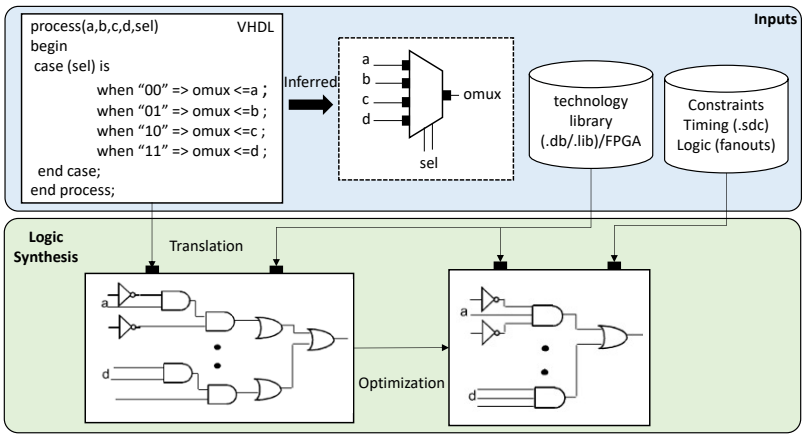



**Figure 2.11:** Logic flow overview.

Each logic synthesis tool has different ways to synthesize RTL, often using their own *commands* to read, analyze and synthesize the description and to write out the gate netlist and the different report files. In the appendix we give examples of logic synthesis scripts for different LS tools like Synopsys Design Compiler (ASIC) and AMD/Xilinx Vivado and Intel's Quartus II for your reference.

### 2.5.7 Technology Mapping

The technology mapping stage takes the optimized gate netlist generated as input and maps it to hardware resources in the FPGA. Basically mapping multipliers to DSP macros, memories to BlockRAM and the rest of the logic to individual LUTs. For this, the technology mapper has to partition the gate netlist into groups of gates that have the same number of inputs as the FPGA LUTs and that generate a single output.

Technology Mapping Example 1: Let's look at some technology mapping examples.

Figure 2.12(a) shows an example of a gate netlist composed of two AND gates and a NAND gate connected to a D flip-flop. This gate netlist has 4 primary inputs (A,B,C and D) and a single primary output (O).

Figure 2.12(b) shows how this gate netlist fits perfectly into a single logic cell composed of a single 4-input LUT and a flip-flop. In particular, the combinational logic is mapped to the LUT, while the flip-flop to the logic cell flip-flop.

Technology Mapping Example 2: This second example shows how to map a larger gate netlist to an FPGA including how the different LUTs are routed together through the FPGAs switch box.

Figure 2.13 shows the gate netlist to be mapped onto the FPGA. In this case the netlist has 8 inputs and a single output. Because of the number of primary inputs, the entire netlist cannot be mapped onto a single LUT, but has to be split into different LUTs. The figure shows the mapping where mapper has grouped the gates based in three different non-overlapping groups based on two conditions: Condition 1: Each group has at most 4 inputs. Condition 2: The group of gates have a single output.

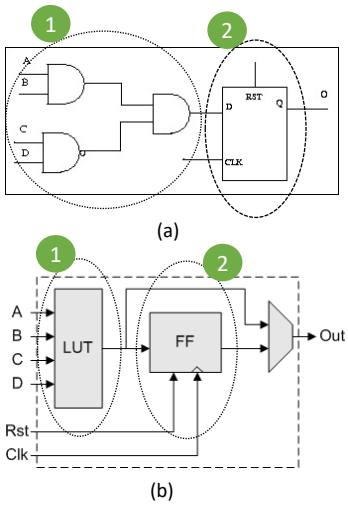



**Figure 2.12:** Technology mapping example.

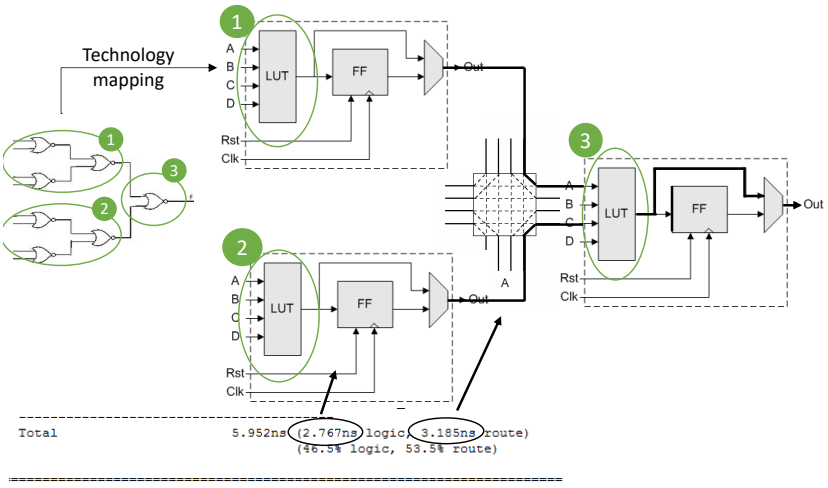



**Figure 2.13:** Technology mapping example of larger gate netlist.

In this example, group 1 is mapped to the first LUT, group 2 to the second and group 3 to a third LUT. This example also shows how the configurable switch box between logic cells is used to connect the outputs of the two first LUTs to the third LUT.

This flexibility is one of the main characteristics of FPGAs. The output of any LUT can be routed to any other LUT through the configurable switch boxes. This flexibility though is also one of the main Achilles heels of FPGAs. Figure 2.13 also shows part of the timing report, reported by the FPGA tools, showing the critical path (longest delay).

As shown, it can be decomposed into two portions, the delay due to the logic mapped onto the LUTs, and the delay due to the routing circuitry. In this particular example, the first is 2.76 ns, while the latter is 3.185 ns, which implies that the delay due to the configurable routing resources is larger than that of the logic. The total critical path in this example is 5.952 ns, which implies that the maximum frequency of this circuit is $f_{max}$=1/5.952 ns =168 MHz.

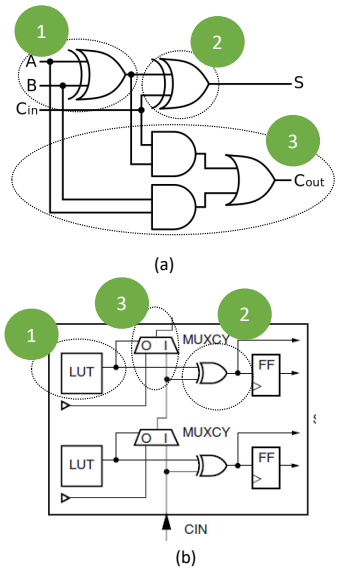



**Figure 2.14:** Technology mapping example of a full-adder. (a) Gate netlist of full-adder. (b) FPGA slices with technology mapping.

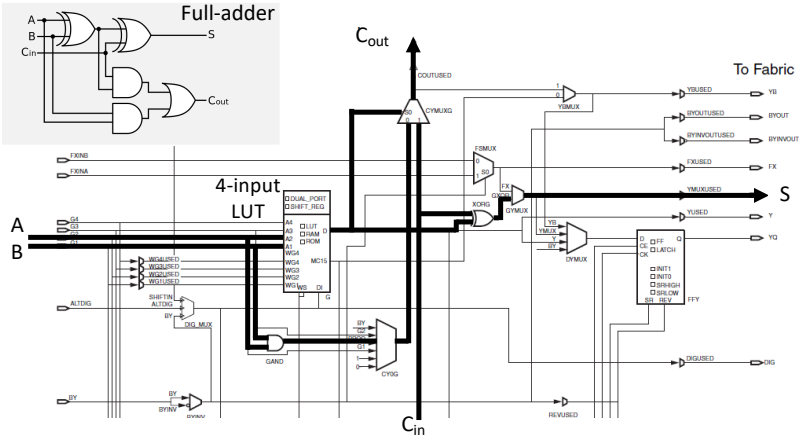



**Figure 2.15:** Detail technology mapping example of a full-adder.

Technology Mapping Example 3 - Full-Adder: One of the most important hardware circuits are adders, as they appear virtually everywhere. The main problem was that full-adders have three inputs (A, B and Cin) and two outputs (S, Cout), but logic cells can only output a single output. This meant that a single full-adder would require two logic cells. One to compute the sum (S) and the other to compute the carry (Cout). This in turn meant that an 8-bit adder would require 16 logic cells or a 32-bit adder 64 logic cells. Moreover, ripple carry adders are notorious for being the smallest type of adders, but also the slowest due to the carry having to propagate through all the full-adders.

To address this, FPGA vendors tweaked their architecture to include a fast carry-chain logic that had two purposes: (1) Speed up the carry propagation, and (2) implement the carry logic in the logic cell such that a single logic cell was needed to map a single full-adder.

Figure 2.14 shows an example of how a entire 1-bit full-adder is mapped onto a single logic cell, with Figure 2.14(a) showing the full-adder and Figure 2.14(b) the underlying FPGA structure. In this case a slice composed of two logic cells. It shows the carry chain traversing the logic cell from the bottom to the top of the slice. It also shows how the logic cells have an additional XOR gate which is used to for the second XOR gate in the full-adder that generates the sum (S).

Figure 2.14 only shows a high-level view of the technology mapping. Figure 2.15 shows the detailed view of a Xilinx Virtex 4 logic cell and how the full-adder is mapped onto the logic cell making use of all the internal resources including the 4-input LUT and the fast carry chain.

In this case the flip-flop is not use, as the full-adder is purely combinational logic.

### 2.5.8 Place and Route

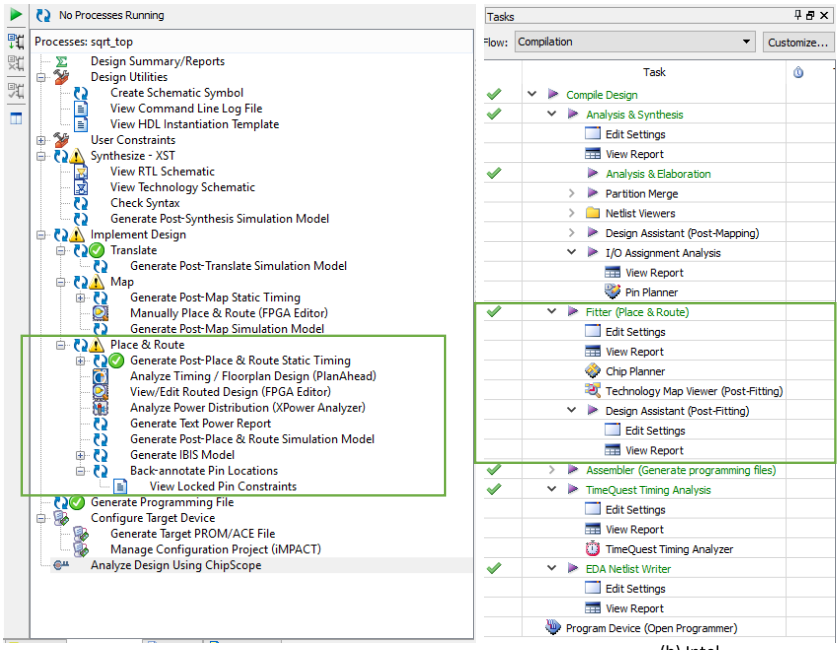
Place and route is the last main stage in the FPGA synthesis process. Once the gate netlist has been mapped to FPGA resources, the place and route stage has to determine where to place these and how to connect them.

This process will place the FPGA resources who's outputs are needed by other resources close together because as shown in Figure 2.13 the

routing delay will significantly impact the critical path delay and hence, the maximum frequency at which the FPGA can run.

Both FPGA vendors have similar flows and tools that guide the user in this process and also allow them to modify the generated placement. For this they provide separate tools like AMD/Xilinx providing PlanAhead and Intel Chip Planner.

Figure 2.16 shows a screenshot of the place and route stages of the two FPGA vendors.




(a) AMD/Xilinx ISE (b) Intel

**Figure 2.16:** Place and Route stages of FPGA vendors.

The output of this stage is the mapped, placed and routed hardware design that started from the RTL code generated from HLS or manually written.

### 2.5.9 Bitstream Generation

This last step encodes all the information generated in the place and route stage and generates a bitstream that configures the individual configuration transistors in the FPGA such that it executes the desired logic function. Because most modern FPGAs are SRAM based, this bitstream needs to be downloaded to a PROM (non-volatile memory) such that every time that the SRAM-based FPGA is turned on, the bitstream is loaded onto the FPGA to configure it.

### 2.5.10 Coarse-grain Runtime Reconfigurable Arrays (CGRAs)

Although the fine-grained LUT-based FPGAs are dominant these days, there are also other alternative architectures. One example is the Coarse-grained reconfigurable Arrays (CGRAs). These are coarse grained FPGAs that consists of arrays of 8-16-bit ALUs that can be configured similarly to traditional FPGAs. Because of the larger granularity the bitstream to

reconfigure them is much smaller than the bitstream required in the fine-grained FPGA cases making them often runtime reconfigurable. This runtime reconfiguration can take less than 1ns and hence leading to being able to be reconfigured every single clock cycle. Thus, these architectures are also often called Coarse-grained **runtime** reconfigurable Arrays (CGRRAs).

There are some commercial products of these devices like Renesas Electronics Stream Transpose Processor (STP) [2] and Samsung's Ultra-low Power SRP CGRA [3]. These CGRAs are mostly commercialized as FPGA IP blocks rather than stand-alone FPGAs to be used in heterogenous SoC onto which to offload particular computationally intensive applications.

Figure 2.17 shows an example of an SoC with a CGRA next to the CPU. This CGRA executes some of the applications that are amenable to hardware acceleration while the rest is executed by the CPU.

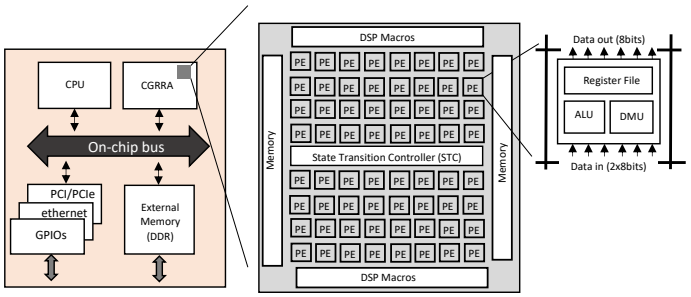



**Figure 2.17:** CGRA IP in heterogeneous SoC.

As shown in the figure the CGRRA is instantiated as one more block in the SoC. The architecture itself is composed of an array of processing elements (ALUs).

### 2.5.11 Renesas Electronics STP

This subsection deals in particular with Renesas Electronics STP as this CGRRA is programmed using HLS.

The STP is composed of tiles onto which the datapath of an application is mapped. Each tile in turn consists of an array of 8×8 Processing Elements (PEs) . Each tile is also surrounded by embedded memory and dedicated DSP blocks. A state transition controller (STC) that is located in the center of the STP is responsible for reconfiguring the tiles with the correct functionality every clock cycle. This reconfiguration takes only 1 ns.

Each PE contains an 8-bit arithmetic logic unit (ALU) , an 8-bit data manipulation unit (DMU) for 1-bit logic operations and 8-bit shifting and masking and an 8-bit flip-flop unit (FFU).

One of the most distinctive features of the STP is how it is configured, as shown in Figure 2.18. Because the design to be mapped onto the CGRRA fabric is divided into individual contexts that are loaded at runtime every clock cycle onto the fabric, it makes sense to start from a sequential description instead of an already parallelized description describe at the

RT-Level (e.g. Verilog). The STP therefore takes ANSI-C as its input and performs HLS to parallelize it.

The scheduling step assigns different portions of the application onto individual control step, based on the available resources. Each control step is therefore synthesized as a context in the STP. The contexts are then mapped onto the STP's architecture, placed and routed. On the other hand, the code for the State transition Controller (STC) is generated with the configuration information for each of the contexts. The context information includes the functionality of each PE and their routing. The loading time for each context takes less than 1 ns. The typical architecture after HLS is an FSM and a data path, where the FSM generates the control signals for the data path. In the case of the STP, each FSM state is a context which is loaded every clock cycle onto the device. The latency of the circuit therefore determines the number of contexts and vice versa and the maximum fabric size is determined by the context which makes uses of the maximum number of PEs.

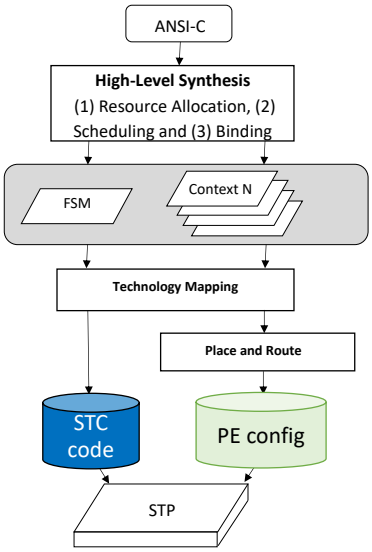



**Figure 2.18:** CGRA configuration flow.

Figure 2.19 shows an example of the code snippet that we have used throughout the book to describe how HLS is used to configure the STP. In this example scheduler generates a 3-clock cycle schedule due to the limited number of FUs allowed (1 adder and 1 multiplier). Every scheduling step is therefore mapped to an individual context in the STP that is reconfigured every clock cycle. In the first context the single addition that adds A and B (A+B) is performed. Then the CGRRA fabric is reconfigured to perform the addition of B and C (B+C) and the multiplication of the previous results with D. The last context reconfigures the STP fabric again to do the very last addition.

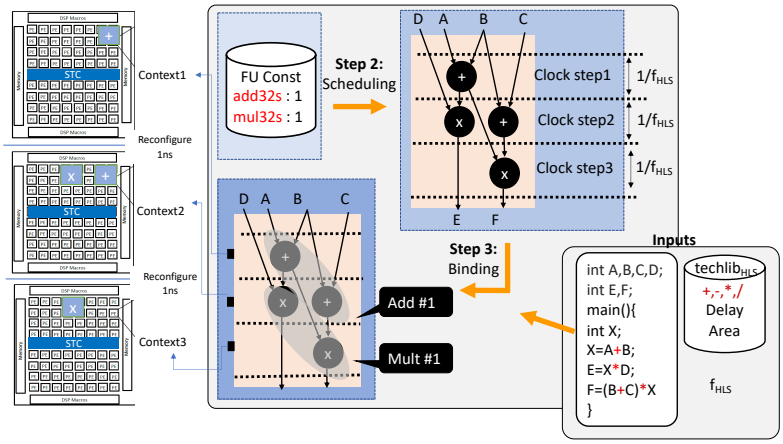



**Figure 2.19:** HLS to configure CGRRAs.

The ability to time-multiplex the hardware allows to save significant silicon area and hence, cost. The main drawback compared to traditional FPGAs is that the reconfiguration process can slow down the execution and also consume more power as the CGRRA requires to reconfigure the PEs every single clock cycle.

Table 2.1 summarizes some of the main differences between traditional fine-grain FPGAs and CGRRAs. It is important to understand their main trade-offs in order to understand when to use one or the other.

**Table 2.1:** Main differences between fine-grain and coarse-grain FPGAs

| FPGA | CGRAA |
|---|---|
| Faster as architecture is static | Slower as it requires reconfiguration |
| Lower power | Higher power due to reconfiguration |
| Larger area overhead | Smaller due to time-multiplexing of HW |
| Larger bitstream | Smaller bitstream (smaller memory) |

### 2.5.12 Summary

In this chapter we have highlighted the different hardware platforms that can be targeted when building dedicated hardware circuits. These are basically the main targets for HLS. In particular ASICs and FPGAs. We have reviewed their main differences and highlighted when to use one over the other. We have also reviewed the design flow to map an RTL description onto an FPGA. This RTL can be manually generated or generated through HLS.

Summary

- ASICs and FPGAs have very different economical and performance/power trade-offs.
- FPGAs are now large enough to map entire complex systems on them, including even embedded processors.
- FPGAs contain heterogeneous resources: LUTS, reonfigurable fabric, BlockRAM and DSP macros.
- The FPGA synthesis flow includes: Logic synthesis, technology mapping, place and route and bitstream generation.

# High-Level Synthesis 3



## 3.1 High-Level Synthesis: How does it work?

The process of converting an untimed behavioral description into structural synthesizable RTL code has been well studied since the 80's. The complete process can be seen in Figure 3.1 composed of three main stages.

**Stage 1: Front-end:** is the synthesizer front-end. Its main purpose is to check for syntactical errors in the behavioral description and to perform technology independent optimizations. These are common optimization done by most software compilers like constant propagations, dead code elimination, common subexpression elimination, etc. These optimizations are extremely important because the final hardware circuit might be much larger than necessary if the input description is not optimized. For example, if the code includes (a+b) - (a+b)/8 the synthesizer does not require two adders to compute (a+b) twice as this will not change.

The output of the font-end is typical in intermediate representation of the optimized behavioral description in the form of a Control Data Flow Graph (CDFG). This CDFG contain the type of operations and their dependencies in graph format.

**Stage 2: Main Synthesis Steps:** This stage is the core of the HLS process and can be further decomposed into three main steps: (i) Resource allocation , (ii) Scheduling , and (iii) Binding .

The **resource allocation** step extracts the resources in the technology library ($techlib_{HLS}$) that will be required to synthesize the behavioral description and stores it in a functional unit resource constraint file (FCNT). The **scheduling** step then times the behavioral description based on the available resources in the FCNT file, the target synthesis frequency ($f_{HLS}$) and the technology library. Based on the target synthesis frequency and the delay of the individual operators, the same behavioral description will be scheduled into different control/clock steps. This is another advantage of HLS. The same behavioral description is automatically retimed for different target synthesis frequencies and technology libraries, without the need to change the behavioral description. This has the additional benefit that the designer can easily switch between targeting an ASIC and FPGA and quickly evaluate the implementation differences. Finally, the **binding** step determines which operation in the scheduled CDFG is executed onto which of the FUs in the FCNT file. These steps will be described in detailed in the next subsections.

**Stage 3: Back-End:** The back-end stage takes the scheduled and bound CDFG as an input and generates structural RTL code in either Verilog or VHDL as output. The RTL structure generated typical consists of a controller in the form of a Finite State Machine (FSM) and a datapath where the FUs reside. These FUs include, adders, multipliers and dividers of different bit widths.

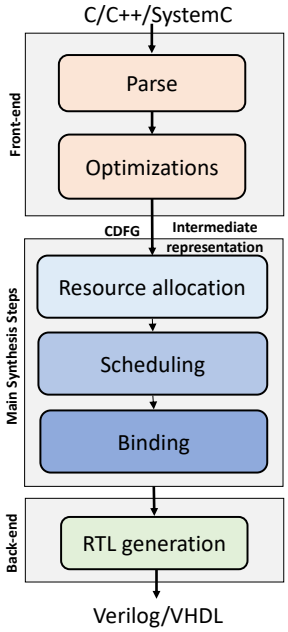



**Figure 3.1:** Main High-Level Synthesis stages: (i) Front-end, (ii) main synthesis steps and (iii) back-end

## 3.2 HLS Main Steps: Resource allocation, scheduling and binding

Figure 3.2 shows an example of a simple ANSI-C program to be synthesized. In this case the program does two multiplications and two additions taken the variables A,B,C,D as inputs and generates two outputs, E and F. In addition to the behavioral description we can also see the two other inputs required by the HLS process. The technology library ($techlib_{HLS}$) and the target synthesis frequency, $f_{HLS}$.

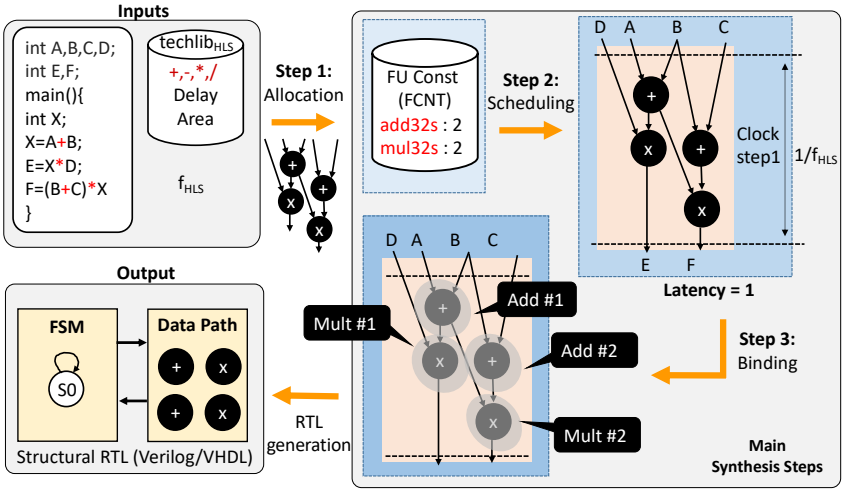



**Figure 3.2:** HLS main steps example: Allocation, Scheduling and Binding.

As shown in the figure, the behavioral description is first parsed by the front-end and the CDFG generated. The next step in most HLS processes is the resource allocation step. This is because most HLS tools perform what is called a resource constrained HLS. In other words, the user specifies how many resources (e.g., functional units) it allows the synthesizer tries to generate the fastest possible circuit given those constraints. Another approach could be to do the exact opposite. Specify the desired performance, e.g., in latency (clock steps required to generate a new output) and the synthesizer would use as many hardware resources as needed to achieve this. The main steps involved in the HLS process are described as follows:

**Resource allocation:** The HLS process parses the CDFG and extracts from the technology library ($techlib_{HLS}$) the number and type of operators (FUs) that it needs to execute the operations in the CDFG. The output is thus, a FU constraint file (FCNT) that contains the number and type of FUs. In this particular example the FCNT file includes two 32-bit signed adders and two 32-bit signed multipliers. The main reason for requiring 32-bit FUs is that the variables are declared as integer types (int A,B,C,D). This severely impacts the quality (area and performance) of the generated hardware and thus, it is extremely important to specify the smallest acceptable bit width. It is not the same to have 32-bit adders and multipliers are much larger and have larger delays than 8-bit or 16-bit adders and multipliers. One problem that we can already see is that ANSI-C or C++ only allow to use standard data types, like char, short int, or int, but to fully optimize the hardware circuit, we often need to specify arbitrary bit widths, e.g., 12-bits or 18-bits. To address this, all commercial HLS vendors provide their own custom data types or use

SystemC which has been standardize by the IEEE and has its own data types. I will cover this in Chapter 7.

**Scheduling:** This second step parallelizes the CDFG assigning individual parts of it to individual clock steps based on the given constraints. These constraints are the FCNT file generated at the resource allocation stage and the target synthesis frequency, $f_{HLS}$. Scheduling is one of the most important parts of the HLS process and much work has been done in the past to find an optimal schedule, although scheduling has been shown to be in general intractable. Because of this, many heuristics that trade-off the quality of the results vs. the time complexity of the scheduler have been presented in the past. Scheduling basically answers the question of how to assign the computations of a program into the hardware time steps.

These scheduling algorithms can be broadly classified into timing-constrained and resource-constrained scheduling as mentioned in the resource allocation sub-section. A popular timing-constraint scheduling heuristic algorithm is force-directed scheduling [4] which is based on a constructive heuristic that iteratively tries to minimize the *force* of the scheduled operations to balance the computations on the given clock steps in order to minimize the number of hardware resources used. Most modern HLS synthesizers perform resource-constraint scheduling, where the number of FU are first constraints and the goal is to find the fastest possible circuit. One of the most popular heuristic is list scheduling [5], where operations are sorted in a list according to a given priority function and then scheduled in the sorted order in the next available clock step. More recently a more elegant way to formulate the scheduling program as a Integer Linear Program (ILP) through a sum of difference constraint (SDC). This can then be passed to an ILP solver which can return the optimal solution [6].

In the example shown in Figure 3.2, the scheduler is able to fit all of the operations in a single clock cycle (clock step 1) by chaining all of the FUs together. Chaining is basically connecting multiplier combinational FUs together between flip-flops. This is only possible if the target synthesis period ($1/f_{HLS}$) is larger than the delay of the chained operations. In this case this leads to a schedule that has a single clock step and hence the entire circuit can be executed in a single clock step. The latency =1 (it takes 1 clock cycle to generate a new results).

In the next subsections we describe how some of the scheduling algorithms work. In particular ASAP, ALAP and SDC scheduling.

**As Soon As Possible (ASAP) Scheduling:** One of the easiest way to schedule the CDFG is using an ASAP technique. This technique basically schedules every operation as soon as it is possible. One of the advantages of the ASAP algorithm is that it leads to the optimal solution if the scheduler is allowed to use an infinite number of resources (basically no resource constraint).

ASAP scheduling maps operations to their earliest possible start time without violating the precedence constraint. Here is a list of the main characteristics of ASAP scheduling:

1. East and fast to compute. Scheduling can be extremely complex and time consuming. ASAP is very fast.

2. ASAP scheduling does not attempt to optimize the number of resources used.
3. Gives the shortest (fastest) possible schedule if unlimited amount of resources is available.
4. Gives an upper bound of the execution time.

Figure 3.3 shows an example of how ASAP scheduling would schedule the code snippet used in this chapter. As shown the scheduler takes as input the parsed code and the FU constraint file (FCNT). In this example the scheduler is only allowed to use 2 adders and 1 multiplier. The figure shows a possible scheduling results based on the delay of the FUs and the selected target synthesis frequency. In this first case ($f_{HLS}$=high the multiplier needs to be scheduled in a separate clock step and hence the final latency of the schedule is 3 clock cycles. If we reduce the target synthesis frequency enough ($f_{HLS}$=low) then the scheduler would be able to chain the adders with the first multiplication leading to a circuit with latency of 2 clock cycles. This further highlights the importance of the accuracy of HLS library characterizer as well as specifying the correct HLS synthesis frequency.

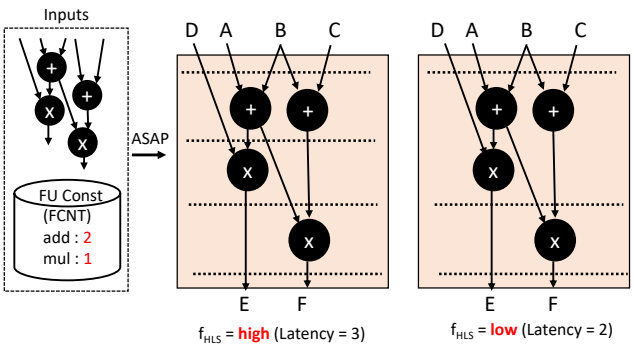



**Figure 3.3:** Example of ASAP scheduling for two different HLS frequencies.

**As Late As Possible (ALAP) Scheduling:** In ALAP scheduling the operations are mapped to their latest possible start time not violating the precedence constraints.

Here is a list of the main characteristics of ALAP scheduling:

1. Easy and fast to compute
2. Finds the longest path in a directed acyclic graph
3. Does not attempt to optimize the resource cost similar to ASAP.

Figure 3.4 shows an example of how ALAP scheduling works for the same code snippet used in the ASAP example with the same FU constraints and using again two different HLS synthesis frequencies. From the figure we can observe that the second addition is now schedule in a later clock step. The latency, similar to the ASAP case depends on the target synthesis frequency.

**Note:** In this particular example, the latencies for the final schedule in the ASAP and ALAP cases match, but this does not have to be the case.

Combining ASAP and ALAP scheduling is often used in other scheduling techniques to determine the 'mobility' of the operators. Mobility is basically determining valid the clock steps for the different operators.

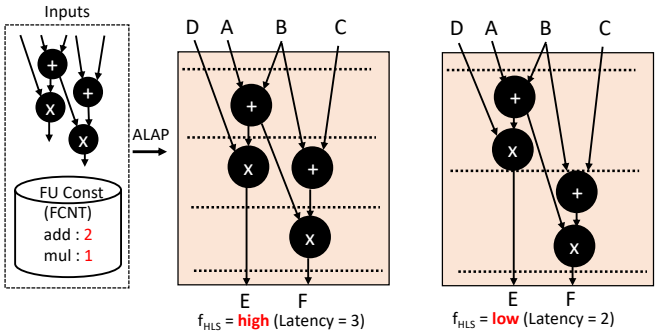



**Figure 3.4:** Example of ALAP scheduling for two different HLS frequencies.

ASAP returns the earliest clock step, while ALAP the latest. Any clock step in between should also be a valid solution.

**Sum of Difference Constraint (SDC) Scheduling:** In SDC scheduling the scheduling problem is formulated as a mathematical optimization problem that is then solved through a solver. More specifically as in integer linear problem (ILP). This makes it possible to solve the schedule formulation in polynomial time (fast). The formulation is basically a linear objective function with linear constraints (e.g.,==,<=, >=).

SDC scheduling was introduced by Zhang et al. [6] and is the scheduling method used in some commercial HLS tools like Vivado HLx and SmartHLS.

In SDC scheduling every operation in the DFG is assigned a variable. The dependencies between the different operations (variables) are then formulated as equations, hence the name difference constraints.

Here is a list of the main characteristics of the SDC scheduling technique:

1. Optimizes the latency often leading to the optimal results and better results than previous methods.
2. Fast solution as can be solved in polynomial time.

Figure 3.5 shows a step-by-step example of SDC scheduling works for the same code snippet used in the ASAP and ALAP scheduling examples. To better explain this scheduling method, in this case we use $f_{HLS}$=10ns and we also define the delays of the adders (6ns) and multipliers (9ns). The SDC scheduler follows these steps, also shown in the figure as follows:

**Step1: Assign variable to every operator:** Assign each operation in the DFG a variable name. In this example the first adder is $X_{add1}$, the second adder $X_{add2}$, and the multipliers are $X_{mul1}$ and $X_{mul2}$.

**Step2: Dependencies constraints:** Model the dependencies in the DFG mathematically as difference constraints equations.

**Step3: Handling clock constraints:** Update the difference constraint equations based on $f_{HLS}$ and the delay of the different operations.

**Step4: Handling Resource constraints:** Create a topological sorted list of the resource constrained operations. Model the constraint as another difference constraint equation and add it to the previous equations.

**Step5: Formulated cost function:** Formulate the overall cost function that needs to minimize the sum of the variables in the system, subject to all the constraints generated.

**Step6: Run Solver:** Call the solver used to obtain the schedule of each operation (clock step in which is operation is scheduled will be reported if a valid schedule is possible.

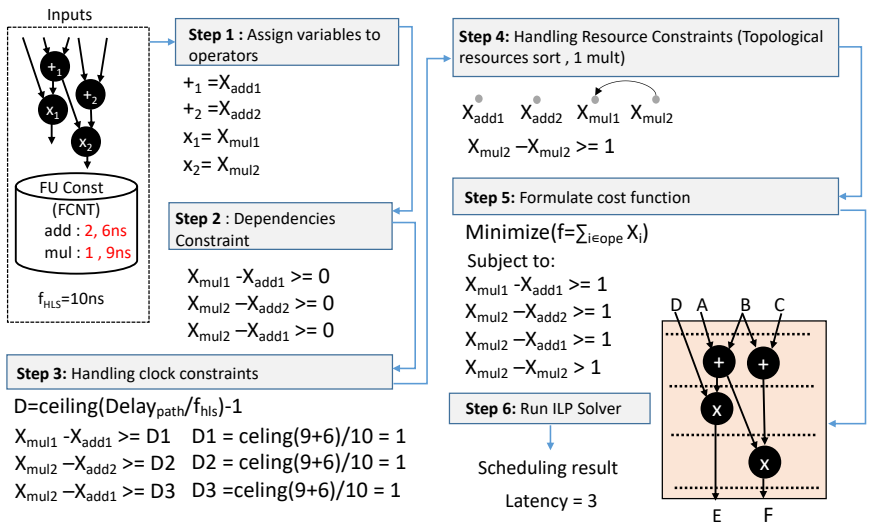



**Figure 3.5:** Example of SDC scheduling for $f_{HLS}$=10ns.

**Note:** The SDC scheduler cannot find the optimal solution because there are many different possible topological sorting. Hence, the solution cannot be guaranteed to be optimal.

---

**Example:** Re-do the dependency constraint equations if the FCNT is updated to only have 1 adder and 1 multiplier.

**Solution:**
$X_{mul1}$ -$X_{add1}$ >= 1
$X_{mul2}$ –$X_{add2}$ >= 1
$X_{mul2}$ –$X_{add1}$ >= 1
$X_{mul2}$ –$X_{mul2}$ >= 1
$X_{add2}$ –$X_{add1}$ >= 1 // new constraint equation

In this case a new constraint needs to be added in the step 4 due.

---

Here is another example when we lower the frequency constraint to 10Mhz=100ns.

**Example:** Re-do the same example now when the target synthesis frequency is lowered to 10Mhz (100ns). Re-formulate the dependency constraint questions if the FCNT now allows 2 adders and 2 multipliers. What would be the latency of the final schedule?

**Solution:**
$X_{mul1}$ -$X_{add1}$ >= 0
$X_{mul2}$ –$X_{add2}$ >= 0
$X_{mul2}$ –$X_{add1}$ >= 0

In this case no resource constraint are needed. The final latency of the circuit would be 1 clock cycle. All the operations can be scheduled in a

single clock step.

---

**Binding:** This last step assigns the different operations in the scheduled CDFG to the different FUs in the FCNT file. In this case because we have two adders and two multipliers, the binding stage has to decide which adder and multiplier will execute which operation. Although this might seem trivial, the binder needs to make sure that the operations across the entire scheduled circuit are balanced as these FUs will be shared across multiple operations which requires multiplexers.

The synthesis results in this example is also shown in the figure. Typically this is structural RTL (Verilog or VHDL) code composed of a controller (FSM) and a data path. The FSM in this case has a single state (S0) as the scheduled circuit has a latency of 1 and the data path contains the two 32-bit adders and multipliers.

One of the main advantages of HLS is that it allows to generates different hardware circuits by simply changing the HLS constraints. For example in Figure 3.6 we have changed the FCNT from two adders and multipliers to a single adder and single multiplier. In this case the scheduler cannot schedule all four operations in the CDFG in a single clock cycle because as shown in the FCNT file it only has one 32-bit adder and one 32-bit multiplier.

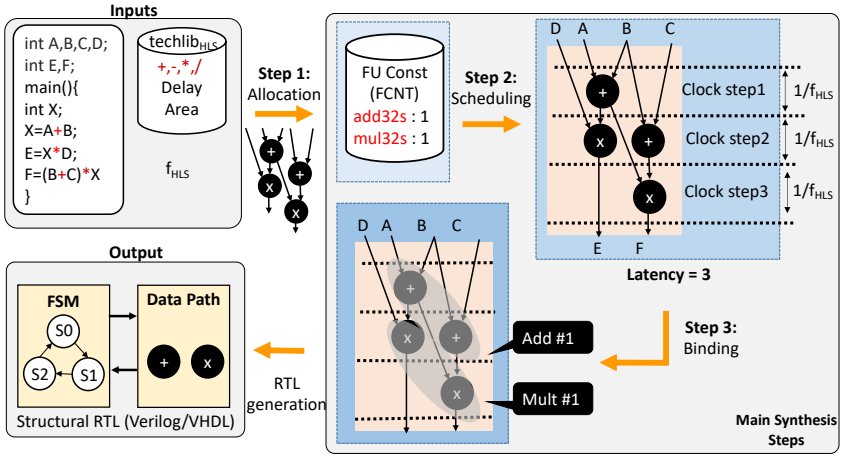



**Figure 3.6:** Example of main HLS steps when the FCNT is changed.

In this case the scheduling step requires three clock steps to map the operations in the CDFG without violating the dependency constraints as well as the resource constraints specified in the FCNT file. This leads to a new circuit with a latency of 3 clock cycles. The output of Figure 3.6 shows that the FSM now has three states (S0, S1 and S2) and the data path now only contains one adder and one multiplier.

The circuits shown in Figure 3.2 and Figure 3.6 are functional equivalent, but the hardware circuit completely different.

## 3.3 Back-end: RTL Generation

The last step in the HLS process is the generation of the RTL code. Figure 3.7 shows the typical structure of the resultant circuit, for the two scheduling results presented in the previously in this chapter. The

typical circuit has a FSM and a data path. The data path contains the FUs generated in the resource allocation stage. Because these FUs are shared for different operations in the code (at least in the example of latency=3), muxes are inserted before and after. An FSM is also generated to generate the control signals for these muxes in order to steer the data across the data path accordingly.

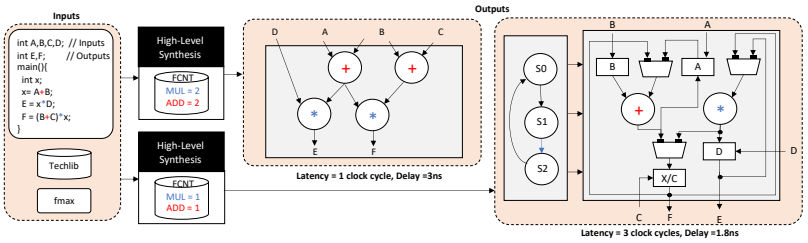



**Figure 3.7:** Typical circuit generated by HLS. FSM+Datapath if the latency is larger than 1.

It is interesting to observe that in the circuit with latency=3, not only the FUs are shared, but also the registers. As shown one register holds at different state the value of variable X and C. This is one key benefit of HLS. It is very efficient sharing resources (FUs and registers).

In the example with latency=1, no FSM is needed because the circuit has a latency of 1 clock cycle. This is very rare because most larger circuits will take more than 1 clock cycles to generate the output.

We observed in the past that when hardware designers have to build their circuits, they often tend to over-design them to avoid resource sharing. In the example shown they would prefer building the circuit with latency =1 instead of the slower, albeit smaller version. The reason for this is that the design and in particular the verification process is much more complex due to the FSM.

Although this is the typical hardware circuit generated by HLS, modern HLS tools also allow to fully pipeline the circuit. In this case no FSM is required as data is *streamed* through the data path. The rate at which the data is streamed is called the Data Initiation Interval (DII) which has to be specified by the user. I will cover this detal when reviewing HLS options in Chapter 5.

## 3.4 HLS Library Generator

As shown throughout this chapter, the quality of the generated hardware circuit deeply depends on the delay estimates of the different FUs. The question that you might ask yourself is how these delays are estimated.

ASIC style HLS tools all include a library generator that needs to be executed once before using the HLS tool in order to re-build the HLS technology library for the target technology at which the IC will be manufactured. It is important that the technology matches. If not, it can lead to serious timing issues and hence, timing closure problems.

Figure 3.8 shows an overview of how these library generators work. They are typically separated applications to which the user specifies the

logic synthesis tool and also the technology library (.lib/.db). The library generator then automatically generates the RTL code for individual FUs of different bit withs (e.g., 1-bit adder, 2-bit adder, 4-bit adder) and synthesizes this RTL with the given target technology library. For this it generates the synthesis script for the particular logic synthesis tools specified. The bit-width step between the different FUs can often also be configured. Normally HLS tools will interpolate/extrapolate the delay and are of FUs with bit widths not available in $techlib_{HLS}$.

This process continues until all of the components in the library generator have been synthesized and their area and delay extracted. Finally the library generator outputs the $techlib_{HLS}$ in the particular format for that HLS tool. Unfortunately, there is no standard for HLS technology libraries similar to .lib or .db files in logic synthesis. Thus, every HLS tool has its own format.

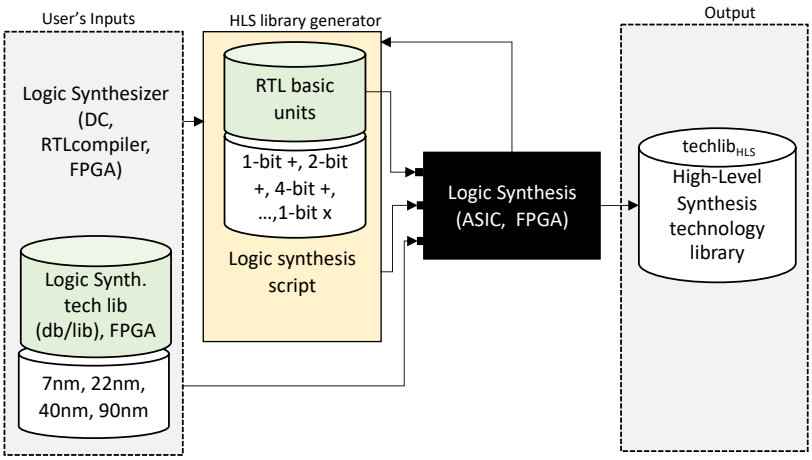



**Figure 3.8:** HLS library generator overview.

You may have used HLS tools from FPGA vendors and these do not include a library generator. You might wonder why? The reason is simple. FPGA vendors do have a library generator in-house, but they do not need to release it as they only need to configure the $techlib_{HLS}$ for their FPGA families once and include these libraries in their HLS tools. ASIC vendors need to provide the library characterizer as ASIC technology libraries are not publicly available and different HLS users will tape-out using different technologies of different foundries.

**Functional Unit Chaining:** One additional thing to consider is that HLS is very good at chaining operations in a single clock cycle. This significantly reduced the number of scheduling steps required. The problem here is that the delay of multiple chained FUs is not the same as the sum of the individual FUs delay. Logic synthesis tools do a very good job optimizing chained FUs. Hence, the library generator will also need to generate different FU chaining configuration and add the delay and area to $techlib_{HLS}$. This step must be repeated for unsigned and signed FUs.

FU chaining as mentioned is very common in HLS. It is important that the delayed of chained FUs is accurately estimated by the HLS library characterizers.

Figure 3.9 shows three different cases. Figure 3.9(a) is the FU (in this case an adder of bit width 'bw') for which we want to find the delay.

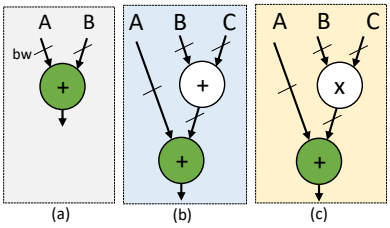



**Figure 3.9:** Examples of FUs chaining. (a) Individual FU; (b) Two identical FUs chained; (c) Two different FUs chained

**Table 3.1:** Individual vs. chaining delay examples (45nm Nangate Opensource).

| | Delay | Description |
|---|---|---|
| add8u | 3.0 ns | stand-alone |
| | 0.4 ns | chained from add16u |
| | 0.0 ns | chained from add32u |
| | 0.0 ns | chained from mul8u |
| add16u | 5.4 ns | stand-alone |
| | 4.9 ns | chained from add16u |
| | 0.0 ns | chained from add32u |
| | 0.0 ns | chained from mul12u |
| mul8u | 12.8 ns | stand-alone |
| | 8.8 ns | chained from mul8u |
| | 0.0 ns | chained from mul32u |
| | 5.9 ns | chained from ad32u |
| mul16u | 26.5 ns | stand-alone |
| | 12.6 ns | chained from mul16u |
| | 0.0 ns | chained from mul32u |
| | 19.7 ns | chained from ad32u |

Figure 3.9(b) shows a possible configuration when the adder is chained to another adder. Figure 3.9(c) shows another scenario when the adder is chained to a different type of FUs, a multiplier in this case.

Table 3.1 shows how the delay of different unsigned adder and a multiplier of different bit widths changes when synthesized separately vs. chained to another FUs from different bit widths. We can observe that the delay when chained to larger FUs gets reduced to 0, further highlighting the importance of capturing the FUs chaining effect in the $techlib_{HLS}$.

This finding obviously mean that the HLS library generator will take much longer to run as it has to generate all of the different chaining configurations, synthesize them and extract their delay and area.

To further highlight the effect of chaining on the scheduling result, and hence, final hardware circuit see the example below.

---

**Example:** Figure 3.10 shows an example of a loop that accumulates 8 numbers. It also shows two possible schedules based on the target synthesis frequency specified (b) 100MHz(=10ns) and (c) 50Mhz(=20ns) considering that the adders have all a delay of 7 ns. In the 50MHz case the scheduler can chain all of the adders together although the sum of the individual adders would violate the frequency constraint of 20ns (7 ns+7 ns+7ns= 21ns). A naïve HLS tool with a poor library generator would schedule this circuit using two clock cycles. In this example $techlib_{HLS}$ also shows the delay of three adders chained together equal to 18ns, thus, allowing the scheduler to chain all of the adders in a single clock cycles, leading to a much faster circuit.

Figure 3.10 also shows how different target synthesis frequencies and FUs delay affect the performance and also the area of the resultant circuit, further highlighting the importance of the library generator on the quality of results.

---

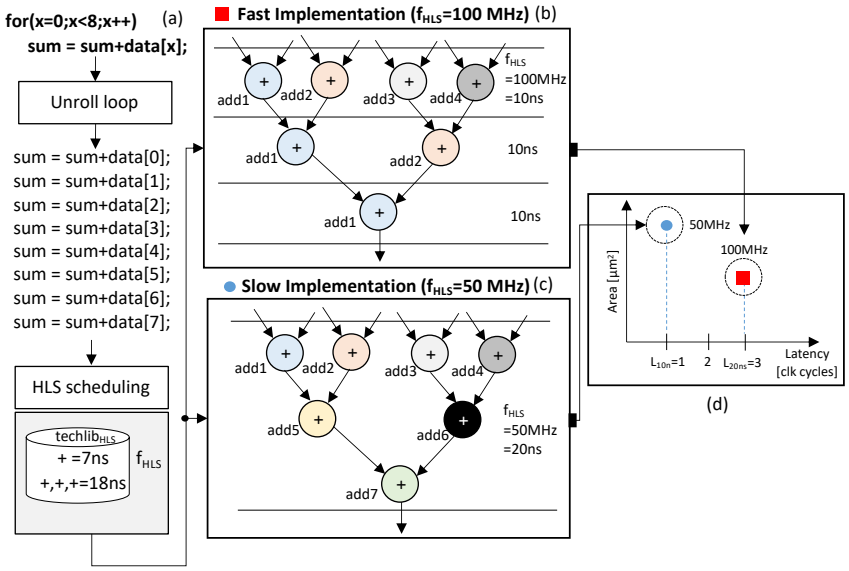



**Figure 3.10:** Example of the importance of the library generator on the quality of the synthesized circuit.

## 3.5 Conclusions

This chapter has covered the main process HLS steps. It is extremely important to fully understand these steps as HLS tools will follow these steps and anyone using HLS will only be able to generate high-quality RTL if he/she knows what the synthesizer is doing, why, and how to control it to generate the desired RTL.

**Summary**

- High-Level Synthesis is the process of converting behavioral descriptions into efficient RTL.
- HLS is composed of three main steps: Resource allocation, scheduling and binding.
- The circuit generated by HLS contains a controller (FSM) and a data path.
- HLS is very good sharing resources (FUs and registers) leading to possibly smaller circuits and manually generated. • The quality of the HW circuit generated from HLS highly depends on the accurate pre-characterization of the different FUs including the chaining effect.

## 3.6 Solved Problems

**Scheduling**

Consider the following ANSI-C code that takes as input an array of eight, 8-bit numbers and returns the sum of all the 8 values. A synthesis directive to full unroll the loop has been specified.

**Listing 3.1**: accumulate.c

```
1
2 int accumulate(char data[8]){
3 int x, sum;
4 // pragma unroll=all
```

```
for(x=0;x<8;x++)
        sum += data[x];
return(sum);
}
```

(i) Draw the DFG of the generated circuit assuming that the HLS tool can use as many FUs as needed to fully unroll the loop. Annotate next to each FU their bit width.

(ii) Schedule the DFG considering the following delays : 8-bit adder=2.0ns, 9-bit adders=2.1ns, 10-bit adders=2.2ns (ignore any chaining effect), for the following target HLS synthesis frequencies $f_{HLS}$=100Mhz, 200Mhz, 400Mhz and 800MHz. Discuss which results is the best in terms of latency and throughput.

**Solution**

(i) Figure 3.11 shows the results of the parsed DFG. A tree-height reduction optimization is shown that allows to fully unroll the loop, while reducing the input to output delay.

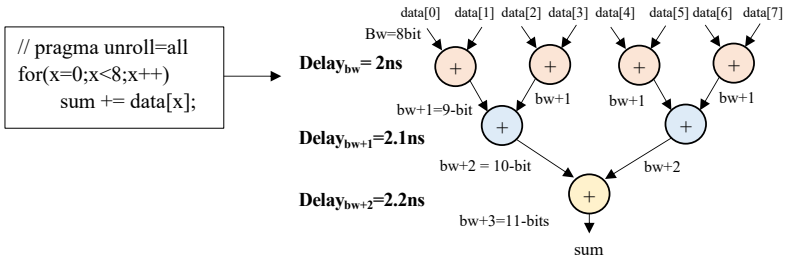



**Figure 3.11:** DFG with tree-height reduction optimization of unrolled loop in accumulator example

(ii) Figure 3.12 shows the different scheduling results for each of the target synthesis frequencies, and table 3.2 summarizes the performance results in terms of latencies and throughput.

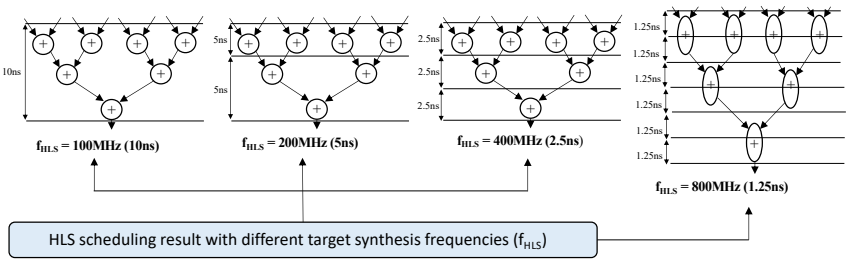



**Figure 3.12:** Scheduling result for accumulator of 8 numbers for $f_{HLS}$=100MHz, 200Mhz, 400Mhz and 800MHz

From the results we can observe that the latency increases when the target synthesis frequency increases as the scheduler needs to insert additional clock steps to keep the timing correct. In the 800Mhz case the scheduler needs to schedule additional cycles to allow the adders to finish. This optimization is called multi-cycle operations (it takes every adder two clock cycles to finish).

In terms of throughput, the results for $f_{HLS}$=100MHz and 200MHz are the same as the increase in operating frequency gets cancelled by the additional clock step required by the scheduler. Thus, the circuit operates

| $f_{HLS}$ | **Latency** | **Throughput** |
|---|---|---|
| 100Mhz | 1 clock | 100Mbits/s |
| 200MHz | 2 clocks | 100Mbits/s |
| 400MHz | 3 clocks | 133Mbits/s |
| 800Mhz | 6 clocks | 133Mbits/s |

**Table 3.2:** Latency and throughput results for different scheduling results.

at a clock frequency that is twice as fast, but takes twice as long to generate an output. The same happens for $f_{HLS}$=400MHz and 800MHz.

Try to synthesize this example with your HLS tool of choice and plot the area vs. latency/throughput trade-off curve. Do the results meet your expectations?

# Verification 4



## 4.1 Introduction

Being able to design hardware faster is not very useful if it makes the verification process more complex. Fortunately, HLS also facilitates the verification process significantly. As shown in the introduction, HLS allows to generate fast simulation models that are orders of magnitude faster than RTL simulations. More importantly, it allows to re-use the untimed test vectors used in the pure C/C++ simulations for timed simulations. In this chapter we are going to learn how this is possible.

Figure 4.1 shows a typical HLS verification flow. A designer would start with a behavioral description that has been optimized for SW only. This behavioral description can be compiled with a standard SW compiler like gcc or g++, and is verified using untimed input test vectors. Golden output results created in SW are also used to test the correctness of the SW description.

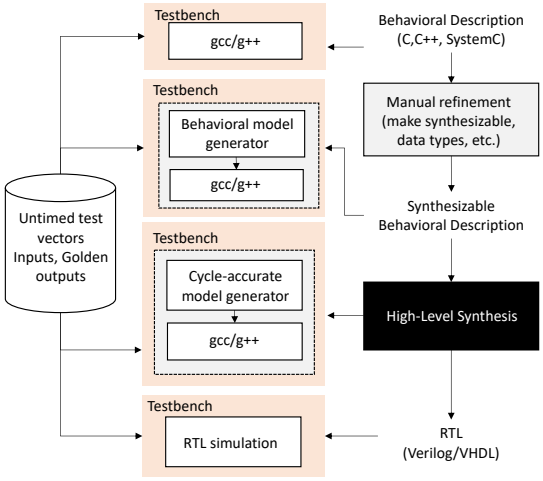



**Figure 4.1:** Typical HLS verification flow.

The HLS designer then needs to refine this SW description and optimize if for HLS. This implies : (1) Make it synthesizable and (2) optimize it for an efficient hardware implementation. Let's look at these optimizations closer:

**Making a behavioral description synthesizable:** Most HLS tools impose limitations to what can be synthesized and what cannot. Some notable limitations include:

1. Recursion : The HLS tools has to be able to statically determine how often a function is called to build the hardware.
2. Dynamic memory allocation: 'Mallocs' are allowed if these are only called once, but similarly to recursion, no memory can be created and deleted during the execution of the program as it is not possible to generate a HW circuit that creates another hardware circuit at runtime.

**Data type refinement:** Optimized hardware often requires to optimize the bit width of the operators in order to obtain the smallest, fastest and lowest power design. Unfortunately, using standard programming languages for HLS only allows to use their standard data types, e.g., char is 8-bits, and int a 32-bit signed variable. Moreover, SW descriptions often make use of floating-point data types. These require larger and more complex functional units.

To address this, HLS vendors provide their own data types to allow specify any arbitrary bit width and also fixed-point data types.

Moreover, HLS vendors also provide HW extensions for their C-subsets. These C-subsets are often called 'C for HW' and include physical data types, e.g., allowing variables to be declared as registers or arrays directly declared as memories. In this case the new C for HW description is not compliant with standard SW languages and thus, cannot be compiled using a regular SW compiler like gcc or g++. The input language chapter deals with this issue in depth.

## 4.2 Behavioral Model Generation

Refining the data types implies that overflow and quantization errors can now be happen in the new description. Thus, the HLS designer needs to re-simulate the newly refined behavioral description to make sure that the output is still correct. A deep numeral analysis is required to make sure that not catastrophic errors are introduced in the system.

To allow HLS users to verify the correctness of their refined synthesizable behavioral description, HLS tools often provide a behavioral model generator. This model generator reads the C for HW and creates a new C/C++ description that models the dedicated bit widths and HW extensions. This simulation model is SW compilable and hence, can be compiled using gcc/g++. This allows the HLS user to verify that the refined description is still functional equivalent to the original behavioral description. In the case that only the data types have been refined using SystemC then g++ can be used without the need of a behavioral model generator. This is mainly because the data types in SystemC are defined as C++ templates, making the language still compilable using g++. This is one of the advantages of using C++. Templates give it a great amount of flexibility that ANSI-C does not have.

## 4.3 Cycle-accurate Simulation

The behavioral model simulations discussed previously allows to only verify the functional correctness of the behavioral description, not the timing as this is still unknown until after HLS, and in particular after the scheduling stage of the HLS process.

To further speed-up the verification process, many HLS tools provide fast cycle-accurate simulation model generators as an alternative to having to do a full RTL simulation.

Figure 4.2 shows an overview of how these cycle-accurate model generators work. They are separate tools within the HLS environment that take as input the scheduling result of HLS process and generate another C/C++ or SystemC program that mimics the behavioral of the HW cycle by cycle. This is possible, because after the scheduling stage the timing is fully known.

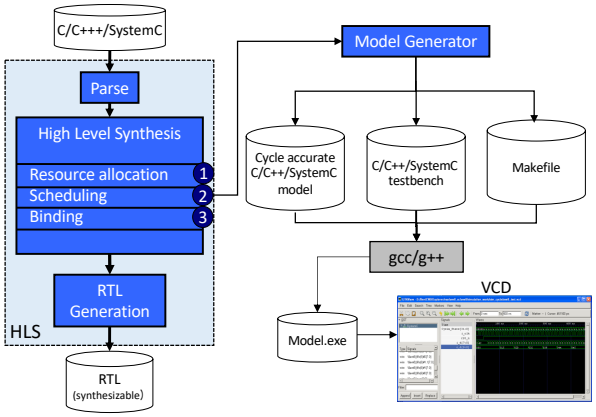



**Figure 4.2:** Cycle-accurate model generation flow overview.

As shown in the figure, the model generator can also create the testbench and the Makefile to compile the model. Because the model is generated in a standard SW language it is then compiled using a standard compiler (e.g., gcc or g++) that can then be executed. These cycle-accurate models can also generate a Value Change Dump (VCD) file that will allow users to study the waveform with the exact timing of the generated hardware circuit.

You may ask yourself how these cycle-accurate models work, and the answer is very simple. Figure 4.3 shows an example of a cycle-accurate model for the simple program used in Chapter 3. The figure shows the HLS scheduling result that is then passed to the cycle-accurate model generator. This generator creates another behavioral description that is basically a large switch-case statement. Every case statement corresponds to one of the scheduling steps and computes the operations scheduled in that particular state. It also reads and writes the primary inputs and outputs in its corresponding case statement like reading A and B in state s0 and outputting the new value E in state s1 and the value of F in s2.

This large switch-case statement is continuously executed until a given exit condition is reached, e.g., number of test vector exhausted.

Using cycle-accurate models instead of a full RTL simulation has multiple advantages:

1. Fast execution. It has been reported that these cycle-accurate simulation models are between 10-100× faster than RTL simulations [1]. One thing that has to be considered though when using these cycle-accurate simulation models is that for larger models the compile time is often non negligible.
2. RTL simulator licenses: These compilable cycle-accurate simulation models only require free SW compilers (e.g., gcc or g++). RTL simulations require a commercial RTL simulator which in turn requires a yearly license fee. Less RTL simulation imply less RTL simulation licenses and hence, cheaper verification costs.

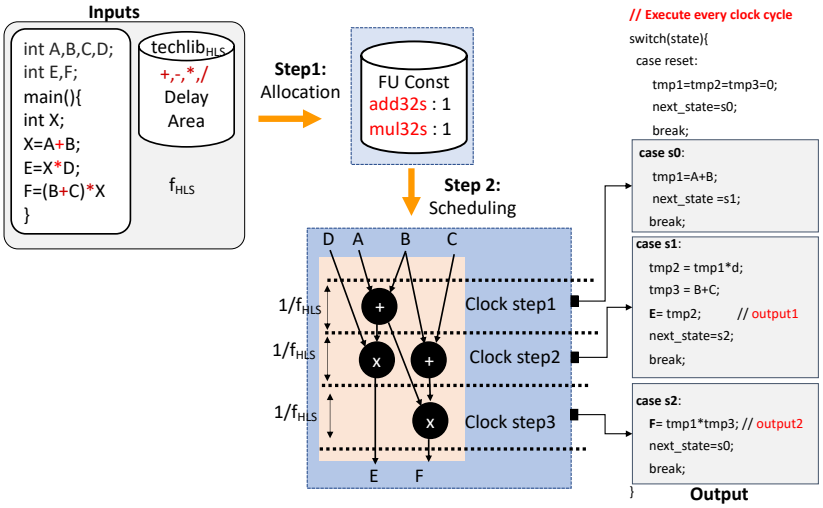



**Figure 4.3:** Cycle-accurate model example.

## 4.4 RTL Simulation

The final verification step involves performing a full RTL simulation of the Verilog or VHDL generated by the HLS process. If the timing was checked using the cycle-accurate simulations, then this should be just a simple sanity check to make sure that the RTL generated is correct, which it should be construction.

The HLS tools also have to generate the script to call the chosen RTL simulator. This is important because there exists a multitude of commercial RTL simulators, i.e., Aldec Active-HDL, Siemens Modelsim, Synopsys VCS, Cadence Incisive, open-source simulators most notably Icarus Verilog as well as the FPGA vendor's own simulators, e.g., AMD/Xilinx isim. Appendix 11 shows different examples of these simulation scripts

The RTL simulation should be functional equivalent to the result of the SW and behavioral simulation and the timing identical than the cycle-accurate simulation.

## 4.5 Testbench

One of the key aspects of HLS is that, as shown in Figure 4.1, the entire verification flow allows to re-use the original untimed test vectors and golden outputs. These are the test vectors used to verify the original SW description. But how is this possible? How can untimed test vectors be used in timed simulations like in the cycle-accurate or RTL simulation cases?

Figure 4.4 shows an example using the same code snippet used previously in this chapter. The top figure shows the hardware module with the primary IOs of the module (PIs=A,B,C and D and POs=E and F). The figure also shows a timing diagram of the circuit showing how the finite state machine generated by the HLS process changes based on the scheduling step. In vvery clock cycle the state changes from s0 to s2. The timing diagram also shows in which states the PIs are being read into the circuit and in which state the POs are written out (s1 of E and S2 for F).

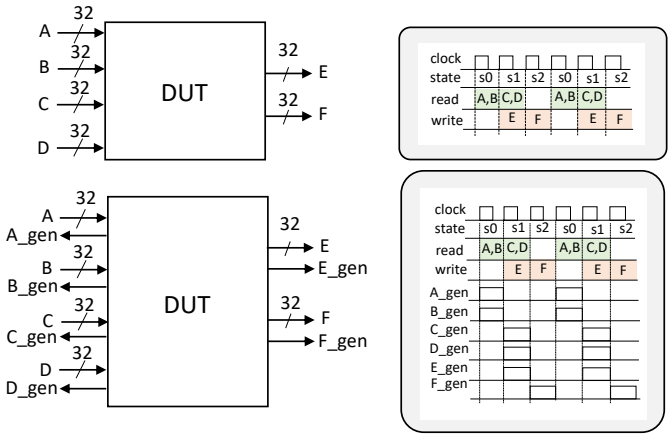



**Figure 4.4:** Example of transactional level simulation using untimed test vectors.

The question is how does the testbench generated by the HLS tool as part of the different simulation model generators know when to pass a new test vector and when to compare a golden output when a new valid output is generated?

One relatively straight forward way to do this is by adding additional control signal to the generated circuit for every primary IO. In Figure 4.4 we can observe that every IO has a new output signal associated with it, e.g., in the case of input A, the control signal generated is A_gen. This control signal indicates that the circuit requires an input for A and is only raised in the state that this happens. In the output case, e.g., E, the control signal E_gen is only raised when a valid output is generated.

The testbench can now use these control signal to pass a new untimed test vector to the specific input only then the control signal is raised and compare a golden output when the control signal of the corresponding output signal is raised. Very simple, but very clever.

It should be noted that this is only intended for functional verification. The final circuit should not have these control signals; thus, the HLS tools provide the option to generate, but also to eliminate these control signal. This is extremely helpful when the timing of the circuit has still not been fixed. Once the timing is fixed the user can remove the control signals and freeze the test vectors. The designer can then insert dummy '0' to the untimed test vectors to make the timing match the final circuit.

Figure 4.5 shows an example of a test vector composed of 5 value (3,4,5,1 and 14). If the synthesized circuit has a latency of 1 it implies that it will read and write an output every clock cycle. Thus, the testbench can fully re-use this test vector file. If the latency changes to two clock cycles, the test vector will need to be updated by inserting a dummy '0' after every valid test vector as the circuit will only read a test vector every other clock cycles. Finally, in the last case, the latency of the circuit is three, which implies that only every third clock cycle a test vector is read, and thus, requiring of two additional dummy '0'.

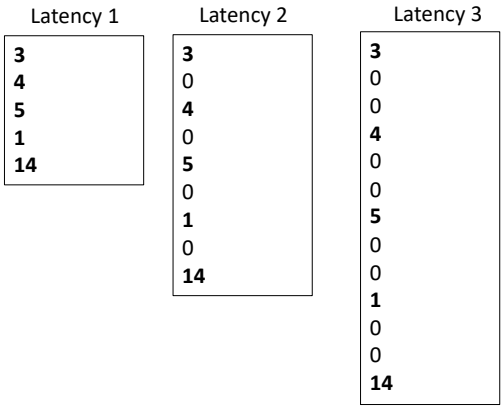



**Figure 4.5:** Adjustment example of untimed test vectors based on latency of synthesized circuit.

## 4.6 Conclusions

This chapter has reviewed the main verification steps in modern HLS flows, including untimed functional verification as well as timed verification through cycle-accurate simulation models and RTL simulations.

**Summary**

- High-Level Synthesis allows to generated different types of simulation models. These simulation models have different speed vs. accuracy trade-offs
- HLS also allows to re-use the untimed test vectors through transactional simulation. One way to accomplish this is by generating control signals for every primary input and output.

# 5 HLS Optimizations



## 5.1 Introduction

Starting the HW design from an untimed behavioral description is like drawing a picture on a new canvas. The painter might have the idea of what to paint in his mind, but not know exactly how to accomplish it.

In HLS, the question is how to generate the desired hardware from an untimed description that does not even have a clock or rest. This is one of the main benefits, but also drawback of HLS. It allows to generate functionally equivalent circuits, but with very different underlining structures and hence, different area, performance and power trade-offs from the same description. This is done by setting different HLS synthesis options. The designer has to fully understand all of these synthesis options and how they interact with each other.

Luckily. HLS vendors provide an arsenal of optimizations that you can use. This chapter will cover the most widely used optimizations. This should enable you to generate the desired hardware circuit. We call these options **synthesis knobs** (can you find them at the book cover?) and classify them into three different groups:

**Global Synthesis Options ($knob_{opts}$):** These are HLS options that are applied to the entire behavioral description to be synthesized. These include how to encode the FSM, synthesis frequency and synthesis mode.

**Local Synthesis Directives ($knob_{pragma}$):** These synthesis directives specify how to synthesize individual operations in the behavioral description. Vendors use pragmas (comments) for these. They allow to e.g., control how to synthesize individual arrays, functions and memories.

**Functional Unit Constraint ($knob_{FUs}$):** This knob allows the HLS user to control how many FUs the final circuit should have. These include adders, multipliers and dividers of different bitwidths.

Setting these knobs to different values will generate hardware circuits with unique area vs. performance trade-offs. Some of these knobs' settings are complementary, while others are overlapping (the same circuit can be obtained by setting different knobs). Chapter 6 deals with the problem of how to set these knobs automatically such that a trade-off curve of Pareto-optimal designs is quickly found, called HLS design space exploration (DSE) . One can imagine that because the number of unique knob settings is very large, the search space is too large to be enumerated exhaustively, and hence, fast heuristics are needed.

It should be noted that every HLS tool has its own set of default options. This implies that the same behavioral description will lead to a very different circuit when synthesized using a different HLS tool. Some tools by default fully parallelize the description, while other tools do not perform any optimization. It is therefore important to understand how

as a HLS user you can generate the smallest possible hardware circuit that meets your performance and power constraints.

## 5.2 HLS Synthesis Knobs

The next sections describe these HLS synthesis knobs in detail, and how they affect the resultant area and performance of the synthesized circuit.

### 5.2.1 Global Synthesis Options ($knob_{opts}$)

These are options normally specified in a synthesis script and apply to the entire behavioral description to be synthesized. Some of these global options include how to synthesize loops, arrays, and functions, similar to the local synthesis directives, but are applicable to all of the loops, arrays, and functions in the behavioral description at the same time. With these global synthesis options, there is no way to specify different options for individual operations. For example, in the case of loops, all loops will be unrolled or not unrolled, but there is no way to unroll some of the loops, pipeline others and not unrolled some other loops in the description. Other options which are complementary to the local synthesis directives include the synthesis mode, clock constraint, data initiation interval (DII) for pipelined designs, encoding scheme for the finite state machine (FSM) controller (one-hot, regular encoding), etc.

### 5.2.2 Local Synthesis Directives ($knob_{attrs}$)

Most commercial HLS tools make extensive use of synthesis directives in the form of pragmas . These pragmas are inserted directly at the behavioral description in the form of comments. This has the main advantage of allowing a very fine controllability over the synthesis result and hence the final micro-architecture. The following code snippet shows a loop that iterates 8 times. The pragma specified in this case before the loop as a comment tells the HLS tool if the loop should be fully unrolled, partially unrolled, not unrolled at all or folded (pipelined).

```
/* pragma unroll= 0 | partial | all | fold */
for(x=0;x < 8;x++)
```

This example highlights how local synthesis directives can be used to control how a loop should be synthesized. The main constructs in the behavioral description that have the highest impact on the final micro-architecture in HLS are loops, arrays and functions. As shown before, loops can be completely unrolled or only partially unrolled. Loops can also be pipelined (folded) with different Data Initiation Intervals (DIIs) . Arrays can be mapped to registers or memories of different types and ports and functions can be inlined or not.

### 5.2.3 Functional Units Constraint ($knob_{fus}$)

The number of Functional Units (FUs) allowed will affect the parallelism that can be extracted from the behavioral description. A well-known optimization technique called resource sharing [7] can be used in order to reduce the area. In resource sharing, a single FU is reused among different computational operations in the behavioral description. For ASICs, the total design area can be significantly reduced as HLS can easily maximize the amount of resource sharing. In contrast, it was also shown that for FPGAs, resource sharing can actually lead to larger designs [8]. This is mainly because the multiplexers required to share individual FUs have shown to require more logic resources than the actual FUs being shared. Commercial ASIC-style HLS tools typically perform resource-constraint HLS. This implies that the first stage in the HLS process is the resource allocation stage, which outputs a functional unit constraint file (FCNT) . This constraint file can in turn be edited to allow the designer to control the amount of resource sharing desired, which in turn leads to a micro-architecture of unique area vs. performance trade-off. This constraint set by the user delimits the maximum number of FUs that the HLS process can use, but does not imply that all of those resources will end up being used, as this constraint only specifies the upper bound of FUs that can be used. Depending on the other two knobs, less adders might be required, but under no circumstances, the HLS tool should use more FUs than specified by the user.

Out of all three knobs, the local synthesis directives knob is the most powerful one as it basically decides upon the overall underlying micro-architecture. Commercial synthesizers also typically give priority to local synthesis directives when global synthesis options with opposing effects have been specified. It should be noted that these knobs are complementary, but also largely overlapping. E.g., a loop will not be fully unrolled if the number of functional units does not allow it. This shows how important it is to fully understand the consequences of limiting the number of FUs when specifying synthesis directives.

The next section will describe the most relevant knobs in more detail.

## 5.3 Synthesis Mode

The synthesis mode is a global synthesis option ($knob_{opts}$) that enables the synthesis of applications with different characteristics. Most HLS tool vendors provide different synthesis **engines** under their hood. Figure 5.1 shows the most common ones and the applications where one or another engine should be used.

Most applications can be classified into one of these three families:

**Data intensive applications:** These are the most common applications used with HLS and include arithmetic intensive applications like DSP and video processing applications. They might also involve computationally intensive applications with little control structures like encryption algorithms, but that have large amounts of parallelism.

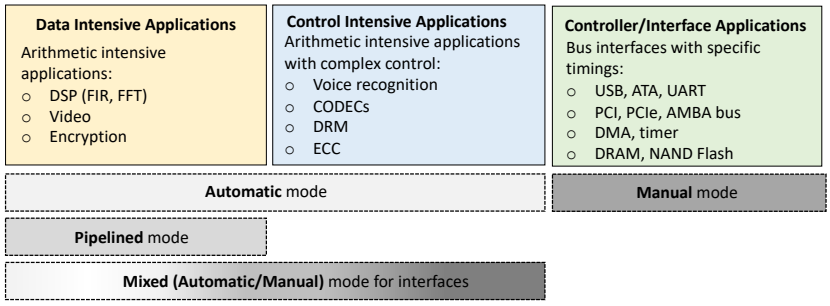



**Figure 5.1:** Overview of different HLS engines applied to different applications.

**Control intensive applications:** These applications are also i arithmetic intensive , but contain also many control structures like nested if-else conditions. In the past, these type of applications were not suitable for HLS, because the HLS tools were not able to extract enough parallelism from these applications to justify a dedicated hardware accelerator. Modern HLS tools make use of complex compiler optimizations like different types of speculations that allows to much better parallelize these applications.

**Controllers or bus interfaces:** This category of applications include bus controllers that have very specific protocols which require specific timing that needs to be satisfied. Some HLS tools cannot synthesize these types of applications as regular C/C++ has no timing notion, while other HLS tools provide extensions in their C languages to deal with this.

HLS vendors might include different HLS engines to deal with all of these applications.

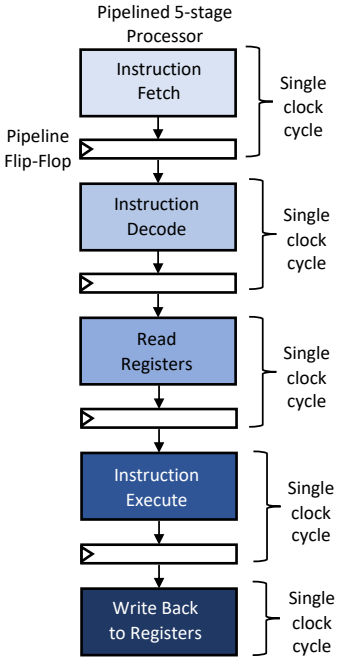



**Figure 5.2:** 5-stage pipelined CPU example

### Automatic Synthesis Mode

Automatic synthesis mode is the traditional HLS synthesis mode covered in chapter 3. It is also what most people identify with HLS. The result of this process is an FSM+Data path where new inputs are read in state X and a new output is written in state Z.

### Pipeline Synthesis Mode

For many multi-media applications throughput is very important. To achieve higher throughputs the hardware has to be fully pipelined. Pipelines function like a factory with multiple different stages allowing data to be fed into the pipeline every clock cycle, or every N clock cycles, based on the Data Initialization Interval (DII) .

Pipelining is a well-known parallelization optimization that allows to parallelize the execution of any application. Figure 5.2 shows a typical 5-stage pipelined CPU architecture. The key that enables pipelining is the pipeline latches that allow to de-couple the different combinational portions of the circuit into unique pipeline stages.

Thus, if throughput is important (e.g., in multimedia applications), then the HLS user might need to fully pipeline the circuit. This requires that the user specifies that the circuit should be pipelined and also the DII. DII basically specifies after how many clock cycles a new input is entering the pipeline. It thus, also determines the interval at which a new output is generated. E.g., if DII=1, then a new input enters the pipeline every

clock cycle and when the pipeline is in steady operation mode, an output is generated every clock cycle too.

Figure 5.3 shows an example of the different synthesis results when synthesizing the code snippet shown in Figure 5.3(a) (used throughout the book) in automatic HLS mode vs. in pipeline mode.

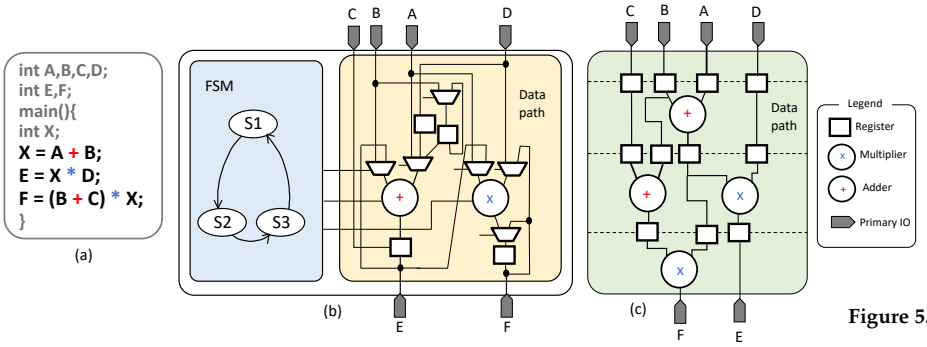



**Figure 5.3:** Pipeline vs. traditional HLS.

In this example, in automatic HLS mode the FCNT is set to 1 adder and 1 multiplier. As shown in Figure 5.3(b), the generated hardware circuit consists of an FSM and a datapath, where the FSM has three states as the circuit has a latency of 3 clock cycles (the schedule has three control steps). When synthesizing the same behavioral description in pipeline mode with a DII=1, the HLS engine will generate the circuit shown in Figure 5.3(c). This circuit also has a latency of 3 clock cycles, but a throughput of 1 output per clock cycle, with both circuits operating at the same clock frequency. The pipelined circuit has no FSM as the datapath is fully spatially laid out and no resources are shared. Thus, the pipelined version requires 2 adders and 2 multipliers. It is not possible to generated a pipelined circuit with DII=1 with only 1 adder and 1 multiplier.

The way to specify how to synthesize the behavioral description (automatic or pipeline mode) is unfortunately tool dependent. Some vendors require users to specify a pragma on the main function, while other vendors have global options for this.

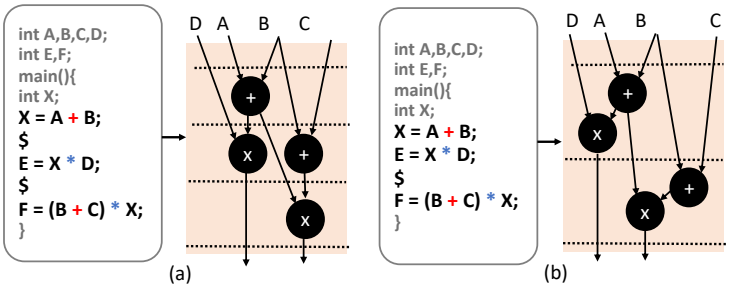



**Figure 5.4:** Manual synthesis mode example. (a) Example of three clock steps. (b) Example with two clock steps.

### Manual Synthesis Mode

Manual synthesis mode involves manually scheduling the behavioral description instead of relying on the HLS engine to do this. Some HLS tools allow to manually schedule the entire behavioral description, while other tools also allow to schedule a portion of the behavioral description, typically the IOs so that the data can be read and written into the hardware module following a particular protocol.

Figure 5.4 shows an example of how this is typically done. In Figure 5.4(a) the source code shown is instrumented with a special symbol, in this case

**$** that represents the clock boundary. The user can with the help of this special symbol specify the clock boundaries and hence, the individual scheduling steps. It is basically the same as buildin an explicit FSM as the final circuit will have as many states as clock boundaries.

As shown in Figure 5.4 (a), this manual schedule leads to a schedule of three clock steps and a final circuit of one single adder and one multiplier. Figure 5.4(b) shows an alternative way to schedule the behavioral description. In this case the schedule has two clock steps, leading to a faster circuit of latency=2.

This gives the user the ability to generate any timings and hence, building any protocol. E.g., this allows to set a given signal to logic '1' during a certain number of clock cycles and then lowering it.

## 5.4 Loop Synthesis

One of the most widely used constructs in any high-level language are loops . These can be synthesized in many different ways. Thus, it is important to understand the effect of these loop synthesis options on the area and performance of the final circuit.

Figure 5.5 shows the different ways to synthesize loops in HLS and their trade-offs in terms of the area of the resultant circuit, the performance (in latency) and also the HLS synthesize time.

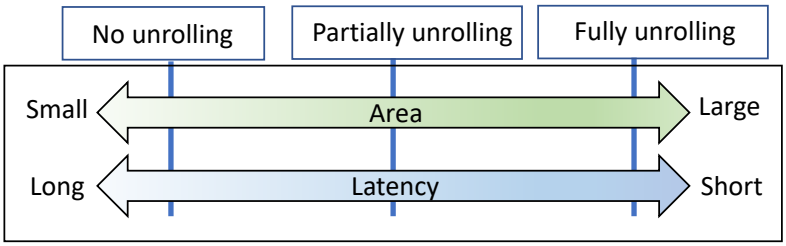



**Figure 5.5:** Loops synthesis trade-offs.

As shown, fully unrolling the loop leads to the fastest of all the circuits, but also the largest, not unrolling the loop implies to sequentially execute each loop iteration like a CPU and hence, leading to a smaller circuit, but slower, while partially unrolling the loop with different unrolling factors lead to intermediate results.

**Global Synthesis Option:** There are two main ways to control how to synthesize loops. A global option will tell the HLS process how to synthesize the loops, but it has very small controllability as the user might want to unroll some loops, while not unrolling some larger loops. Thu

**Local Synthesis Directives:** These directives in the form of comments (pragmas) allow to specifically select how to synthesize individual loops.

**Example 1**: Effect of Loop unrolling on synthesized circuit

The code snippet below computes the moving average of 8 (ave8) numbers. An example of pragma insertion is shown at the loops. The first one

shifts the data of the array that holds the 8 values, while the second loop accumulates the 8 values stored in the array.

```
1  int data[16];
2
3  int ave8(int data_new){
4  int sum=0, x
5  /* pragma unroll=all*/
6  for(x=7;x>0;x--)
7      data[x]=data[x-1];
8  data[0]= data_new;
9  /* pragma unroll=0|=4|=all*/
10 for(x=0;x < 8; x++)
11     sum =sum+data[x];
12 return(sum/8);
13 }
```

Figure 5.6 shows the effect of three different pragma settings of the second loop on the synthesized circuit. In particular, when the loop is fully unrolled (●) ,partially unrolled by a factor of 4 (■) and not unrolled at all (▲).

Figure 5.6 first shows how the HLS parser would modify the behavioral description based on the pragma. In the fully unrolled case, the loop would be completely eliminated, while in the partially unrolled case the loop is expanded based on the unrolling factor. Considering that the HLS tool can use as many FUs as needed and based on the $techlib_{HLS}$ and $f_{HLS}$), the three most likely schedules are shown for each of the three cases.

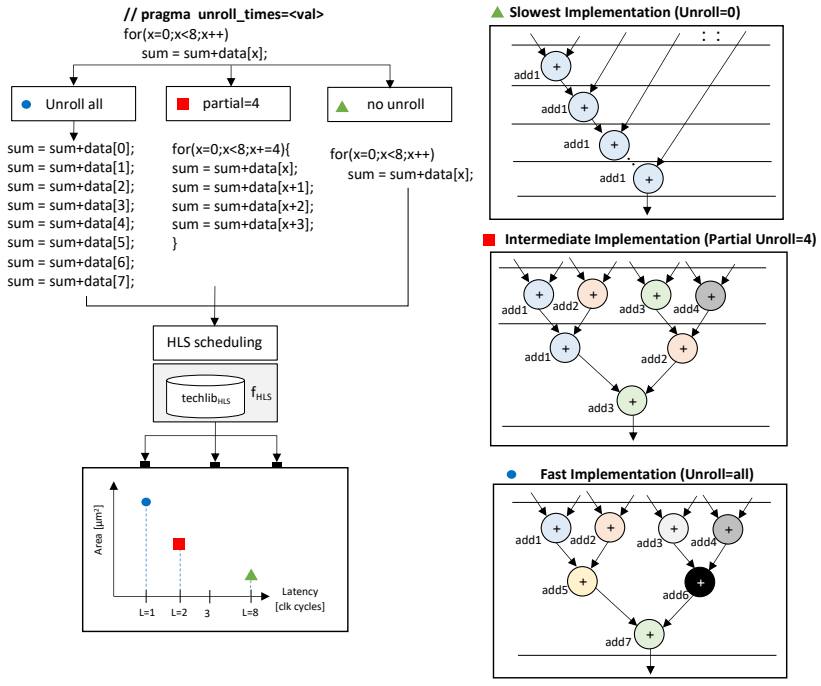



**Figure 5.6:** Example of effect of loop unrolling on average of 8 numbers .

The fully unrolled loop should lead to the fastest, circuit with a latency of 1 cycle, but also the largest as shown in the trade-off graph in the figure because it will use 7 adders. The HLS tool performs a tree-height reduction optimization to minimize the critical path requiring 7 adders. This allows to accumulate the 8 values stored in the array in a single clock cycle.

In the partially unrolled case, the HLS tool will require four distinct adders such that it can add four values stored in the array in a single clock cycle. Three of these adders are then re-used in the second clock cycle to continue with the accumulation of values.

Finally, if the loop is not unrolled, then a single addition is performed every clock cycle, leading to a circuit that has a latency of 8 clock cycles, but only requires a single adder.

These results obviously also depend on how the array is synthesized as it assumes that all of the elements in the array can be accessed in parallel. This is not the case if the array is synthesized as a single port memory. It is therefore extremely important to understand how to synthesize arrays.

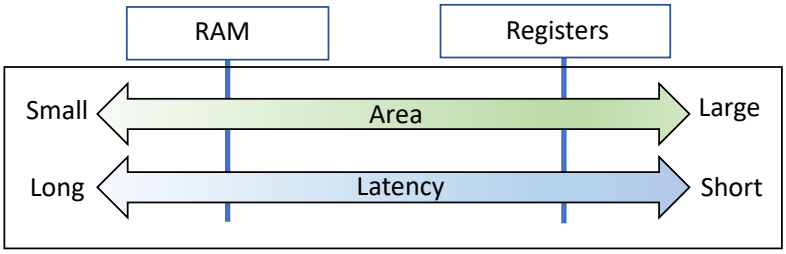



**Figure 5.7:** Array synthesis trade-offs.

## 5.5 Array Synthesis

Arrays are extremely common in any SW program, mainly appearing in conjunction with loops. This makes SW programs so convenient. The problem again for the HLS designer is how to synthesize these arrays. There are multiple options that lead to different trade-offs similarly to loops.

Table 5.1 highlights the most common ways to synthesize arrays classified in whether they are read-only or not. Read-only arrays (arrays that have been initialized and are only read from) can be synthesized as ROMs or pure combinational logic. Read/Write arrays can be synthesized as RAM, registers or be expanded into individual FFs. Many of these synthesis options have sub-options, like e.g., in the RAM case, the user can decide how many ports the memory should have.

**Table 5.1:** Options when synthesizing arrays in HLS.

| Option | Description |
|---|---|
| ROM | Read-only memory |
| Logic | Hard-coded logic for read-only array |
| RAM | Memory, can have multiple ports |
| Register | Register bank with multiple ports |
| Expand | Expand array to individual FFs |

In the case of multi-dimensional arrays HLS tools also offer ways to partition this array in different dimensions, leading to different memory configurations.

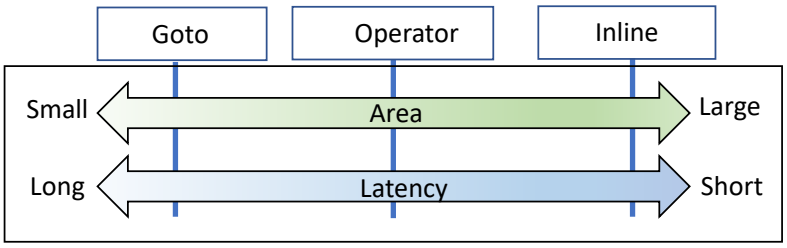



**Figure 5.8:** Functions synthesis trade-offs.

## 5.6 Functions Synthesis

The last major optimization involves synthesizing functions. Figure 5.8 shows the main ways to synthesize them and their trade-offs. Table 5.2 summarizes these three approaches and describes them.

Inlining basically instantiates as separate individual hardware blocks every time that the function is called. Goto does the opposite. A single hardware block is synthesized for that function. Operator on the other hand is a trade-off between both extremes that allows users to control that area vs. performance trade-offs. Basically, the function is encapsulated as a FU and the user can specify in the FU constraint file how many functions it allows the synthesizer to instantiate.

| Option | Description |
|---|---|
| inline | Every function call is synthesized as its own HW block |
| operator | Function is encapsulated as FU. User determines how many. |
| goto | Single HW block for every function call |

**Table 5.2:** Options when synthesizing functions in HLS.

These three options, allows any HLS user to synthesize functions efficiently and to further generated different types of hardware circuits.

## 5.7 Conclusions

This chapter has covered the most common synthesis options that most commercial HLS provide in order to allow the HLS user to generate their desired hardware. It is important that you familiarize yourself with the format of these options provided by the HLS vendors that you are using as the syntax is vendor dependent.

**Summary**

- HLS tools have many synthesis knobs that allow to generate the desired hardware
- These knobs are : (1) Global synthesis options, (2) Local synthesis directives, and (3) number of FUs.
- Arrays, loops and functions appear in most SW description and can be synthesized in many different ways. Learn how!

# High-Level Synthesis Design Space Exploration

# 6

## 6.1 Introduction



As shown in previous chapters, one of the main advantages of HLS is that it allows generating different functionally equivalent hardware circuits from the same design without having to modify the behavioral description. The automated process of generating different designs automatically is called Design Space Exploration (DSE). The way to control the HLS process to generate these different micro-architectures is typically done by setting different synthesis options, also called *knobs*.

We have seen in Chapter 5 that there are three main synthesis optimization knobs.(1) The first knob is global synthesis options that apply to the entire behavioral description to be synthesized. (2) The second exploration knob is synthesis directives added to the source code in the form or comments or pragmas. (3) The last exploration knob allows users to control the number and type of FUs. Reducing the number of FUs forces the HLS process to share resources, which might lead to smaller designs, albeit increasing the designs' latency. It should be noted that setting different family of knobs might have the same effect on the final circuit, e.g., partial unrolling and setting a maximum number of FUs.

The main problem that HLS users face is how to set these knobs in order to obtain a design with the desired characteristics, considering that total number of knob settings is extremely large. Thus, an automated method can find optimal designs is desired, a.k.a. HLS DSE.

HLS design space exploration can be classified as a multi-objective optimization problem (MOOP) as the main goal is to minimize a set of conflicting design parameters. These typically include area ($A$), performance (e.g., given as latency $L$ or throughput $T$) and/or power ($Pwr$), but, as we will describe later, other parameters such as temperature or fault-tolerance could also be included further increasing the search space. Thus, the goal of HLS DSE is not to find a single optimal solution, but a set of designs called Pareto-optimal designs or micro-architectures, which form the Pareto frontier ($\bar{P}$). $\bar{P}$ can be defined as a set of dominating designs $\bar{P} = \{d_1, d_2, \ldots, dn\}$ such that for any design $d_i \in \bar{P}$: $A(d_i) \leq A(d_q)$ and $L(d_i \leq L(d_q)$

In other words, any design $d_i \in \bar{P}$ is Pareto-optimal if no other design in the search space has simultaneously less area ($A$) and less latency ($L$) than $d_i$.

Because the search space is extremely large it is typically not possible to claim Pareto-optimality and hence, the obtained trade-off curve is often referred to as the dominating designs or non-dominated designs. A typical HLS DSE exploration framework is shown in Fig. 6.1, where the explorer is built as a layer between the application and the HLS tool.

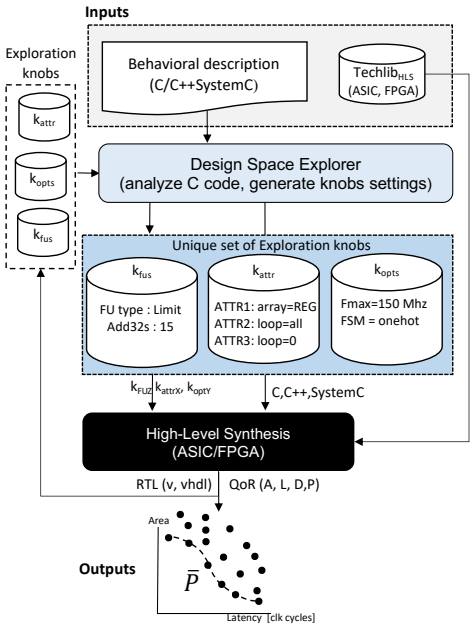



**Figure 6.1:** HLS Design Space Exploration (DSE) overview.

Portions of this chapter were taken from our previously published paper [9].

The main purpose of the explorer is to automatically generate unique sets of knobs and call the HLS tool in order to minimize a given cost function. The HLS tool takes these newly generated knobs, the behavioral description to be synthesized and the technology library and generates the new RTL code with unique design characteristics given in terms of area, latency, delay and power.

### 6.1.1 Exploration Knobs

There are many different ways to obtain unique RTL description from a given untimed behavioral description. Early work on HLS DSE made use of different allocation, scheduling and binding algorithms to obtain different micro-architecture. Since these main steps in HLS are NP-complete and interdependent and the design objectives are in conflict the authors in [10] proposed the use of evolutionary algorithms to explore different binding strategies. Similarly, the authors in [11] use max-min ant colony optimization (ACO) to solve both the time and resource constrained scheduling problems to obtain different designs. Other early work in HLS DSE [12] exploited the unrolling factor of loops and created a flow-based method on the parsed C code using SUIF [13]. The main drawback with this early work is that they assumed that the designer has access to the HLS process, which is not true in practice. Most current work assumes that the HLS process is a black box and set different synthesis knobs before the HLS process is executed. They then invoke the HLS tool and based on the Quality of Results (QoR) reported, generate a new set of knobs. Three main family of knobs exist in commercial HLS tools: (1) Local attributes in the form of pragmas inserted directly at the source code ($k_{attr}$), (2) global synthesis options that affect the entire behavioral description ($k_{opts}$)and (3) number of functional units ($k_{fus}$). Other work explores the bitwidth of functional units in the context of DSP applications to explore the area vs. throughput constraints [14, 15].

Another approach, fully orthogonal to the main exploration knobs just described, is to explore the behavioral compiler phase-ordering. Some HLS tools [16, 17] use as front-end traditional software compilers, and thus, rely on traditional compiler techniques to optimize the program's Intermediate Representation (IR). Thus, the quality of compiler front-end optimizations directly impacts the HLS-generated circuits. Groups of optimizations are often packaged into optimization options , such as -O0 and -O3, for ease. Finding the optimal sequence of optimization phases is an NP-hard problem, and exhaustively evaluating all possible sequences is infeasible. Different optimization options lead to different IRs and hence different type of circuits. Some previous work in this domain includes [18, 19].

### 6.1.2 FPGA vs. ASIC Design Space Exploration

The three knob categories just introduced apply mainly to ASIC-style commercial HLS tools. ASIC designers require a lot of flexibility and controllability when generating their RTL code and hence, all the commercial HLS tools provide a large variety of synthesis options in the form of the knobs described previously. When targeting FPGAs this is not the

case as FPGA vendors mainly target ease of use. Moreover, many HLS optimizations that typically lead to dominating designs when an ASIC is targeted have been shown to have the opposite effect in the FPGA case. E.g., resource sharing, as mentioned before, thus, FPGA vendors' HLS tools do often not allow to control the number of FUs.

Fig. 6.2 shows an example. In this case the exploration results of the average of 16 example shown previously when the number of FUs is reduced when targeting either an ASIC (45nm Nangate technology) or an FPGA (Xilinx Virtex 5) using [20]. Three design points are generated for each target technology: First, using as many FUs as needed to fully unroll the main computational loop. Second, halving the number of FUs, and third, using only one FU of each type. Basically the first configuration does no resource sharing, the second medium and the third maximum resource sharing. As shown in the Figure, the ASIC case leads to a strictly descending monotonic area vs. latency trade-off curve, which implies that all three design configurations are Pareto-optimal, while the FPGA case does not, implying that only the first configuration (no resource sharing) is Pareto-optimal. This effect has been reported and studied in [8, 21] and should hence be taken into consideration.

Moreover, in [22] the authors showed that synthesis attributes that lead to Pareto-optimal designs when targeting an ASIC did not lead to Pareto-optimal designs when targeting an FPGA and proposed a machine learning-based conversation framework that converts the exploration results when targeting one technology to the other automatically.

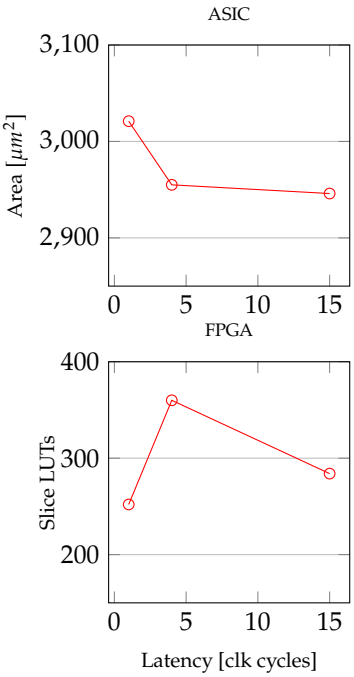



**Figure 6.2:** Average of 16 example FU exploration targeting ASIC and FPGA technology.

### 6.1.3 HLS DSE Benchmarks

There are two benchmark suites specifically dedicated to High-Level Synthesis. CHstone [23] is a benchmark suite containing designs and their testbenches in ANSI-C. ANSI-C is popular when targeting FPGAs as FPGA users are often embedded software engineers that know ANSI-C well. The second benchmarks suite is S2CBench [24], which is written in SystemC. SystemC is preferred by hardware designers as it has hardware extensions including data types (fixed and floating point) and allows to model concurrency. That is why all of the ASIC HLS tools support SystemC. One additional reason for the use of SystemC in ASIC designs is that SystemC has been standardized by the IEEE [25] and also has a synthesizable subset reference manual [26], which allows companies to easily port designs complying with this subset from one tool to another. Moreover, recently the authors in [27] introduced a SystemC/C++ library of commonly used hardware functions and components that can be synthesized by most commercially-available HLS tools.

One other benchmark suite is Spector [28], which contains OpenCL designs. This benchmark suite can also be used for HLS DSE.

Any of these three benchmark suites can be used to measure the effectiveness of the explorer as most academic papers use them to report their exploration results and hence allow to compare the results directly.

### 6.1.4 Exploration Metrics

One very important consideration when doing HLS DSE is to determine what the different design metrics are. Typically these are area and performance. In the ASIC case area is straightforward and is typically measured in $\mu m^2$. In the FPGA case this is not so straight forward. Reconfigurable resources in the form of LUTs, slices or CLBs are typically used, but FPGAs also include embedded hard macros. Most notably DSP macros and embedded memory (BlockRAM). Two different philosophies have been used. The first ignores their usage during DSE as it considers them as a use or loose commodity [29]. The second, builds a global metric and adds these resources to the LUT usage and considers this global metric as total area usage. In terms of performance, there are two metrics that are typically being used. One is latency $L$ given in clock cycles (number of clock cycles from input to output). The second metric is throughput $T$ with $T = f_{max}/L$, where $f_{max}$ is the maximum frequency of the synthesized micro-architecture. Although both metrics are valid, we recommend the use of $L$ because this will not change after logic synthesis and physical design. The $f_{max}$ reported by the HLS tool is not accurate as the wire delay is not considered. Moreover, the generated RTL code specifies adders and multipliers using '+' and '*' symbols and leaves it up to the logic synthesis tool to synthesize them accordingly (e.g. in the case of adders as ripple carry adders, carry save adders or carry lookahead adders). This leads to the first challenge in HLS DSE:

**Challenge:** Can we trust the QoR (Area, delay, latency) reported by the HLS tool to guide the explorer? If the results reported are not correct the Pareto-optimal micro-architectures obtained after HLS will not be the actual Pareto-optimal designs. In [29] the authors showed that in the ASIC case the results could be trusted, but not so in the FPGA case and proposed a method that performs HLS DSE and then synthesizes (logic synthesis) the designs within a certain distance to the Pareto-frontier to find the actual Pareto-optimal designs. Other proposed methods include the use of machine learning to improve the precision of the QoR metrics reported by the HLS tool [30].

### 6.1.5 Design Space Exploration Quality Indicators

The main problem when comparing results from MOOP is how to measure the quality of the results. Several studies can be found in the literature that address this problem in a quantitative manner. Most popular are unary quality measures, i.e. the measure assigns each result a number that reflects a certain quality aspect. Usually a combination of them is used [31],[32]. A multitude of unary indicators exist e.g. average distance to reference set, hypervolume and dominance. Zitzler et. al review of all existing methods in [33]. They also mention that there is not any single indicator able to fully characterize the results. Quality measures are nevertheless necessary in order to compare the outcome of the different DSE methods. The authors in [34] suggest using the following criteria:

*Average Distance from Reference Set (ADRS)*: This measure indicates how close a Pareto-front is to the reference front. Because in most cases it

is not possible to find the actual Pareto frontier, the reference front is obtained by combining the best results of all the methods under consideration. The smaller the ADRS value is, the closer the obtained approximate front is to the reference front. Given a reference Pareto front $\Gamma = \gamma_1 = (a_1, l_1), \gamma_2 = (a_2, l_2), ..., \gamma_n = (a_n, l_n)$ and an approximate Pareto front $\Omega = \omega_1 = (a_1, l_1), \omega_2 = (a_2, l_2), ..., \omega_n = (a_n, l_n)$with a $\in A$ and l $\in L$, where $A$ is the design area and $l$ its correspondent latency. follows:
$ADRS(\Gamma,\Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega)$ where

$$f(\gamma = (a_\gamma, l_\gamma), \omega = (a_\omega, l_\omega)) = max\{|\frac{a_\omega - a_\gamma}{a_\gamma}|, |\frac{l_\omega - l_\gamma}{l_\gamma}|\}.$$

The lower the distance value (ADRS) is, the more similar two Pareto sets are. E.g. a high *ADRS* value tells that an entire region of the reference Pareto-front is missing in the approximation set.

*Hypervolume*: This index measures the hypervolume (HV) of the part of the exploration space that is weekly dominated by the Pareto set to be evaluated. In order to measure this index the exploration space must be bound. We define the bounding point, as the point which has coordinates in the objective space equal to the highest value obtained.

$$HV(\Gamma) = \cup_{\gamma \in \Gamma}\{y \in \mathbb{R}^M, 0 \prec y \prec f(\gamma)\} \tag{6.1}$$

Canonical partial order $\prec$ in $\mathbb{R}^M$ is used to represent that one design is inferior in all objectives to another. The smaller the difference in hypervolume between the heuristic and the reference set, the higher the quality of the result is.

*Pareto Dominance*: This index is equal to the ratio between the total number of Pareto-optimal designs found by the new explorer method also present in the reference Pareto set.

$$Dominance = \frac{\Omega \cap \Gamma}{\Gamma} \tag{6.2}$$

where $\Omega$ is the approximate Pareto-front being evaluated and $\Gamma$ the reference front containing the Pareto-optimal designs. In other words, the number of Pareto-optimal designs found by the new explorer compared to the total number of Pareto-optimal designs. The higher the value the better.

*Cardinality*: The cardinality simply lists the number of dominating designs found. A high cardinality indicates a larger number of solutions to choose from, which should be considered to be positive, although it needs to be interpreted carefully with the rest of the results.

One of the problems when creating a DSE is that for most realistic cases it is not feasible to find the optimal Pareto frontier as the number of combinations is too large to perform an exhaustive search. Thus, the question is how to measure the quality of the exploration results? To compare the quality of multiple heuristics, the dominating designs of all

the heuristic are combined to form a reference front. Then, the quality metrics described previously (ADRS, dominance, etc..) are calculated taking the reference front as baseline. Although this approach does not guarantee to quantify how good the heuristic is compared to the optimal solution, it allows to compare multiple heuristics against each other.

## 6.2 Review of Proposed Design Space Exploration Techniques

Two main categories of HLS DSE techniques have been developed so far. These techniques can be classified into synthesis-based and model based. The synthesis-based methods generate a new set of unique knob settings and call the HLS tool in order to evaluate the effect of these new settings on the quality metrics. This category can be further sub-divided into meta-heuristics like simulated annealing, genetic algorithm and ant colony and dedicated heuristics.

A third category (supervised learning methods) is a mixture between synthesis and model based. The most time-consuming part of the DSE process is the invocation of the HLS tools needs to be invoked. Thus, these techniques use supervised learning to generate predictive models to predict the area and performance given a new knob settings in order to avoid having to invoke the HLS tool. In order to generate these models they first sample the search space by synthesizing a certain number of designs and continuously build a predictive model until the model is stable (the error between the model and the synthesized designs is smaller than a certain maximum error threshold). These methods then continue the exploration using that predictive model instead of invoking the HLS tool. The last category that has been recently introduced is based on static graph analysis techniques. This means that no HLS runs are needed at all. These techniques parse the behavioral description to be analyzed and generate and calibrate offline a set of prediction models that are then used in combination with diverse graph analysis tools. A list of all the different techniques can be found in our previous paper [9].

It is therefore important to understand the strengths and weakness of the different methods which typically trade-off the exploration runtime with the quality of the results.

Fig. 6.3 show a value curve of the different proposed methods. Value curves allow to compare graphically different methods/products, by rating each of them based on different factors. In this case we use runtime, accuracy of the results, how easy is the method to implement, how easy it is to maintain and how easy it is to port to other HLS tools. The y-axis ranks the different methods for a particular attribute in very good (++), good(+), medium (+/-), bad (-), very bad(–).

This classification was mainly done based on our own experience developing different HLS DSE methods [35–39].

From the different curves, it can be observed that the graph-based techniques are the fastest, but also the least accurate, the most difficult to implement, maintain and most difficult to port. This is because they are based on static area and performance prediction models that need

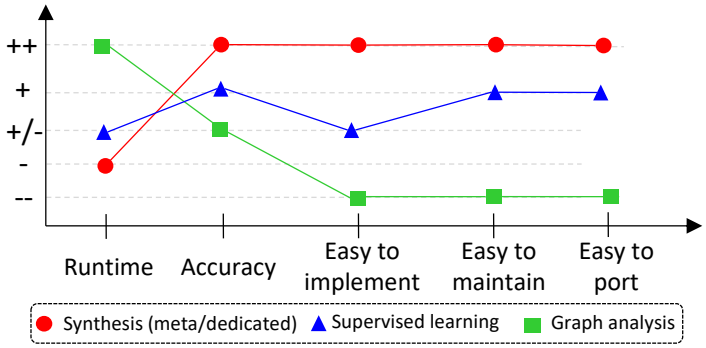



**Figure 6.3:** HLS DSE comparison between different methods.

to be re-adjusted every time the HLS tools is updated or for a new HLS tool. On the other end of the spectrum are the synthesis-based methods (meta-heuristics and dedicated heuristics), which are the most accurate one as for every new knob setting the HLS tool is invoked, but also the slowest for this same reason. These methods are also the fastest to implement (especially the meta-heuristics), easiest to maintain and to port [35, 39, 40]. Finally, the supervised learning based approaches lay between the other two approaches in terms of most of the metrics.

### 6.2.1 Synthesis-based HLS DSE methods

Figure 6.4 shows an overview of a typical synthesis-based HLS design space explorer. This approach requires to generate a new knob settings and then to fully invoke the HLS process to evaluate the effect of this knob on the cost function that drives the explorer. The explorer's goal is to find knob settings that can minimize the cost function. These approaches can be further categorized into meta-heuristics or dedicated heuristics as follows.

**Meta-heuristics:** Meta-heuristics are problem independent heuristics that have shown to lead to very good results for multi-objective optimization problems like this one. Meta-heuristics are often naturally inspired like genetic algorithm, simulated annealing [41] and ant colony optimizations. The authors in [39] make use of simulated annealing for HLS DSE. Genetic algorithm was used in [40] and ant colony in [35]. Other work that makes use of particle swarm was presented in [42], and bacterial foraging optimization algorithm [43] although these works take as input a CDFG and apply different optimizations to directly on the CDFG.

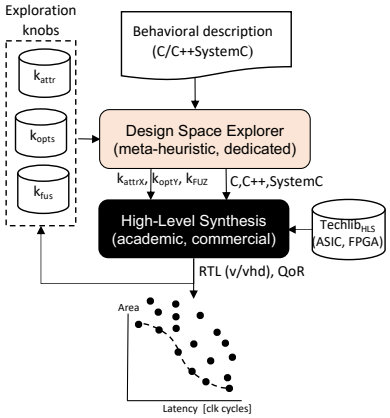



**Figure 6.4:** Synthesis-based HLS DSE.

One of the problems with meta-heuristics is that they depend heavily on the configuration parameters. In the simulated annealing case, e.g. the initial temperature, how fast the temperature should be decreased and how many knobs should be modified in each iteration. In the GA case, how many parents should be generated, what should be the cross-over and mutation rate and after how many generated children should the exploration exit. Moreover, in all cases, the exit criteria needs to be set, e.g. after how many new designs that do not improve the cost function should the explorer move on (either update the cost function or exit). Should these settings be constant for every behavioral description or should they be design dependent?

**Dedicated heuristics:** Due to the uniqueness of the DSE optimization problem, much of the previous work has focused on developing dedicated heuristics. Some initial work focused on speeding up the exploration by applying a set of pre-determined templates to specific pattern in the behavioral description to be explored [36]. Other approaches like divide and conquer where proposed in [44]. In this case loops are explored separately and the exploration results merged in a later phase. The authors in [45] iterative partition the design space into smaller and disjoint subspaces followed by a gradient-based pruning heuristic.

Most applications include loops and arrays. Thus, the authors in [46] and [47] proposed methods to analyze the dependencies between loops and arrays in order to assign the best unroll factors and array synthesis (type and ports). In [48] the authors introduce a method called COSMOS, which performs a HLS DSE as a pre-characterization stage to find optimal configuration in an SoC. Others methods include the partition the of design space based on loop hierarchies [49].

The authors in [50] introduced a heuristic that samples the search space and identifies regions of interest in the design space. The proposed method than iteratively searches for new solutions within such regions. The same authors proposed a different method in [51] that builds a lattice structure of the search space and intelligently explores it. Other approaches use iterative greedy techniques [52].

### 6.2.2 Hybrid: Supervised Learning

In supervised learning, the explorer generates initially a training pool by sampling the search space as shown in Figure 6.5. This implies having to fully synthesize different knob settings. A predictive model for the different quality metrics is then generated by using a predictive machine learning method, which is in turn used to continue with the exploration. This implies that no HLS runs are needed anymore and hence the exploration process can be significantly sped-up. This DSE approach was suggested in similar contexts like the DSE of application-specific multiprocessor SoCs (MPSoCs) where the authors minimize the number of simulations to be executed by combining design of experiments (DoEs) and response surface modeling (RSM) techniques for managing system-level constraints [53]. The authors in [54] applied this technique in the context of dedicated accelerators. In this case the knobs used where high-level design specific parameters like the radix, streaming width, etc. for a DFT.

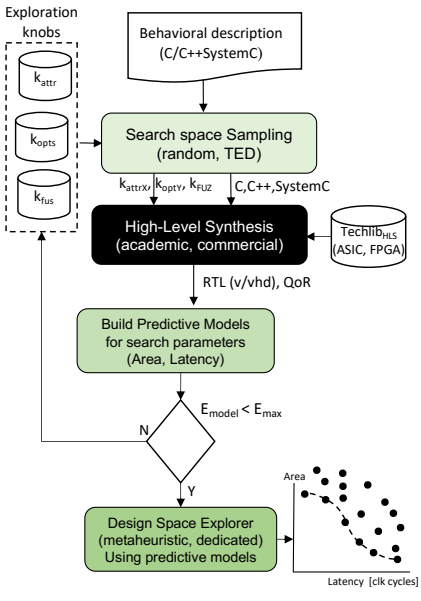



**Figure 6.5:** Predictive model HLS DSE.

The work is again based on the smart sampling of the search space combined with the use regression models to estimate the QoR. In the context of modern HLS DSE addressed in this survey, the authors in [37] introduce a method that randomly samples the search space combined with the mode building (random forest in this case) until the model error is smaller than a given threshold value. The authors in [55] extended this approach and included a more intelligent sampling method based on randomized Transductive Experimental Design (TED). TED judiciously samples representative and hard-to-predict knob settings, and use them for training the learning models. The authors in [56] approach the sampling of the search space by using spectral analysis techniques

for characterizing the solution space and to identify hard-to-predict regions. They then present novel exploration strategy by exploiting this information during the iterative optimization phase based on Response Surface Models (RSMs).

Because of the importance of loops, some supervised learning approaches specialize on predicting the effect of loop unrolling factors on the resultant circuits, by sampling the search space and creating either their own predictive methods [57] or using uses random forest [58].

As shown in Fig. 6.3, supervised learning methods show a good trade-off between runtime and accuracy of results. The main problem is that they are not easy to implement and also a very sensitive to different parameters. In particular the size of the sample set and the type of models uses.

The challenge with supervised learning methods is to determine the size of the training data considering that as shown in [37] the predictive models' error is not monotonically decreasing.

### 6.2.3 Graph Analysis Based

The last category of DSE methods is one that does not require any synthesis at all. Figure 6.6 shows a typical flow. Zhong et al. [59] were one of the first who proposed to perform analytical model estimation for HLS DSE. This initial work was restricted to a small number of knobs (loop unrolling, loop pipelining and array partitioning). The authors in [60] extended this work to deal with a larger number of optimization pragmas. Finally, the authors in [61] target applications given in OpenCL, but based on the highly parallel execution model in OpenCL mainly focus on analyzing the effect of pipelining and parallelism.

These methods are very fast, but are the most difficult to implement, maintain and they might lead to less accurate results. The main reason for this is that they are based on offline-generated models that predict the area and performance of a given behavioral description. This implies that they have to take into consideration the optimizations done by the HLS tools. Moreover, these methods are difficult to port to other HLS tool or even to port to a new version of the same HLS tool. Modern HLS tools are based on multiple heuristics that when modified might lead to different results. These tools also have bugs and limitations that require to be modeled too. For example, past Xilinx SDx versions used register to implement array partitions, whereas the latest version uses BRAM. These implementation details are hard to be captured and maintained in analytical models. Because of this, current work mainly targets FPGAs as the search space is smaller than in the ASIC case.

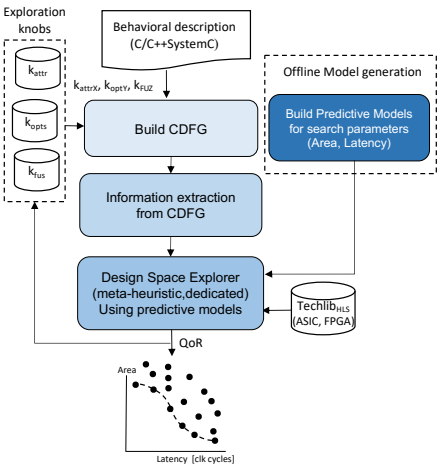



**Figure 6.6:** Graph analysis HLS DSE.

Finally, it should be also noted that even for these methods a search heuristic is still required. E.g. in [60] the target set was to find the fastest one micro-architecture. For this, the authors make use of a greedy algorithm that iteratively tries to reduce the latency between CDFG edges. The obvious advantage of these methods is that due to their speed in evaluating the effect of different knobs' settings, they can explore a much larger portion of the total search space.

The main challenges with graph analysis-based methods are two-fold. First, they also have to model the inaccuracy of the HLS tool, which are

based on heuristics. These methods have to deal with the unpredictability of the HLS tool behavior in order to find the Pareto-optimal regions of the design space. E.g. the authors in [62] highlight the effect of loop unrolling on the controller delay, which is not generated until after HLS. Second, these methods are much harder to maintain as revised versions of the HLS tool require re-calibrating the model.

### 6.2.4 Transfer Learning

A very recent new approach that has been coined as transfer learning based. Transfer learning basically makes use of previously generated exploration results to more efficiently explore new, unseen behavioral descriptions [63–66]. Transfer learning is basically a shortcut to accelerate the exploration process. Previous work always starts with random pragma combinations, and then update them in order to minimize a cost function (either minimize area, latency or a mix between both). In transfer learning the idea is to initialize the pragmas based on results observed form previous HLS DSE exploration results. This allows the explorer to converge faster and hence, lead to the dominating results faster, or allows to evaluate more combinations that have a higher probability of leading to the optimal results.

## 6.3 Building your own HLS Design Space Explorer

This section will teach you how to build your own HLS design space explorer using different simple approaches ranging from a simple exhaustive enumeration of all possible synthesis options combinations to the use of well-known meta-heuristics like simulated annealing (SA) and genetic algorithm (GA). To simplify the problem we will only consider the synthesis directives in the form of pragmas. We have also made available a GUI framework that is able to execute any explorer, display the results and even compare the results of different explorers using the quality metrics discussed previously in this chapter.

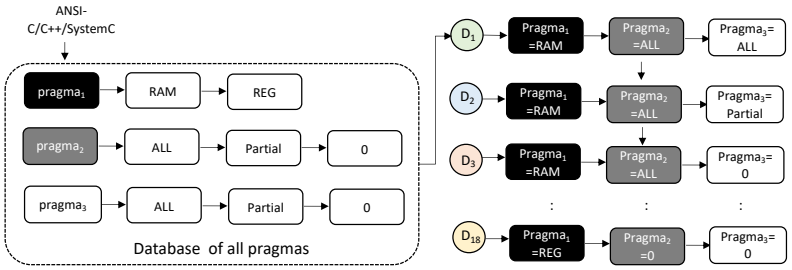



**Figure 6.7:** Exhaustive pragma enumeration DSE.

### 6.3.1 Exhaustive Enumeration

This is probably the easiest way to build a HLS DSE. It basically consists of enumerating all possible pragma combinations, and synthesizing

them all to obtain their area and performance. With all of the designs synthesized it is easy to extract only the Pareto-optimal ones.

Fig. 6.7 shows an example of a design that has three explorable operations. One array and two loops. For simplification we assume that the array can only be synthesized as RAM or Register and the loop fully unrolled, partially unrolled or not unrolled. Thus, in total there are 18 unique combination (2×3×3). The figure shows how each design has a unique pragma combination.

Every new combination is synthesized (HLS) which makes this a very time-consuming exploration method. Thus, more efficient methods are needed for larger benchmarks that have large search spaces.

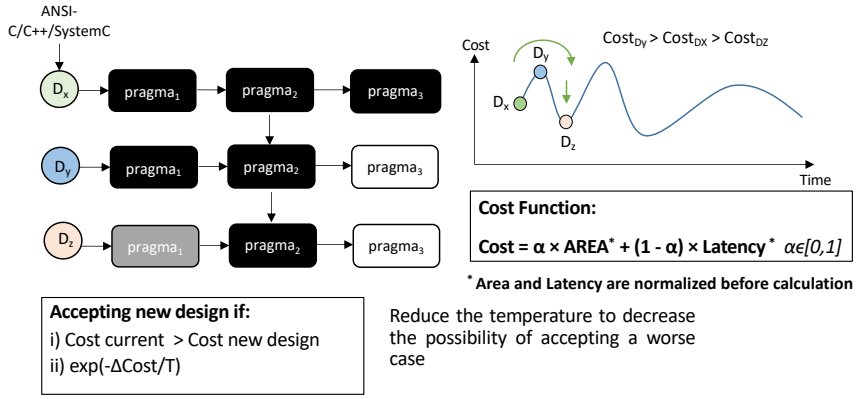



**Figure 6.8:** Simulated Annealing (SA) HLS DSE overview.

### 6.3.2 Simulated Annealer

Simulated Annealing was initially proposed by Kirpatrick et al.[67] as an effective heuristic for multi-objective optimization problems similar to the HLS DSE problem. It is based on an analogy of how metals cool and anneal, but instead of using the energy of a material it uses a cost function. It also requires a virtual temperature (T) that will allow the process to accept new designs with worse cost with the objective to escape potential local minima. Fig. 6.8 shows an example of how to use SA in the context of HLS DSE. The main steps in an SA-based HLS DSE algorithm goes as follows:

**Parse Behavioral description:** Parse the behavioral description, extract explorable operations (arrays, loops and functions) and build a database with all possible pragmas for each operation.

**Initialize Hyper-parameters:** Initialize Temperature (T) to large value and pragma update rate (probability of updating a pragma). Set the cost function weights ($\alpha$ minimize area or $\beta$ latency). These weights determine which part of the Pareto-frontier to explore (high area, high performance or low area, low performance).

**Step 1: Initial random design generation:** Generate a random pragma combination, synthesize (HLS) and compute cost ($C_{current}$).

**Step 2: Generate new design:** Copy the pragmas from the current design and based on the pragma update rate modify only some of synthesis directives values. This update rate is typical small ( 10-20%). Synthesize the design and compute the cost ($C_{new}$).

**Step 3: Keep design or use new:** This step determines if the newly generated design should be used to continue the exploration or if the previous (current) design should be used. For this two criteria are used as shown below:

$$Accept\ new = \begin{cases} Cost_{new} < Cost_{current} \\ Prob_{accept\_new} = exp^{\frac{\Delta_{Cost}}{T}} \end{cases}$$

Basically the new design is used if the cost is smaller than the current design or determined by the Metropolis criteria. This is the main feature of SA. It allows to accept designs with a worse (larger) cost in order to escape from local minima as shown in Fig. 6.8. For this we calculate the probability of accepting the design with higher cost as a function of the cost difference and current temperature. Here T represents the annealing temperature and $\Delta_{Cost}$= $Cost_{new}$-$Cost_{current}$. It can be observed that the probability of accepting a worse solution decreases with the cost difference, but more importantly the probability increases with a higher temperature ($exp^{\Delta_{Cost}/T}$ is closer to 1 with higher T and closer to 0 with lower T). This implies that when the SA explorer starts, the probability of accepting the worst solutions is higher and as it continues and the temperature is reduced, this probability falls. It should be noted that when computing the cost, the area and latency values have to be normalized so that their weights are equal in the cost function. This is done by dividing the area and latency obtained by the largest area and latency obtained so far.
Based on this the new design is either discarded or made the new current design.

**Step 4: Repeat and lower Temperature (T):** Repeat step 2 and step 3 until an exit criteria is reached. This could be e.g., $N$ designs generated that do not lead to a smaller cost. The temperature T is then lowered and steps 1 and 2 repeated until T=0. The temperature can be lowered either linearly or exponentially.

**Step 5: Cost function update:** Update the cost function weights ($\alpha$ and $\beta$) to explore a different part of the trade-off curve and repeat steps 1 to 4.

**Pareto-optimal Design Extraction** Once all the combinations have been synthesized, the Pareto-optimal designs are extracted by comparing the area and performance of all of the designs.

In the example shown in Fig. 6.8 three designs with different pragma combinations are shown: $D_x$, $D_y$, and $D_z$. As shown the Cost of $D_y$ is higher than the cost of $D_x$, but $D_y$ was still accepted such that $D_y$ could be generated from $D_y$, hence, escaping the local minima of $D_x$.

**NOTE:** It should be noted that the most time-consuming part of the exploration process is having to invoke the HLS tool to synthesize the new pragma combination. Because of the randomness of the heuristic, it is possible that the process re-generates previously evaluated pragma combinations. To avoid re-synthesizing previously generated designs, the explorer should store all the previously generated results in a database and check if the a newly generated pragma combination has already been synthesized or not. If it has, it should simply read the values from the database and continue the process.

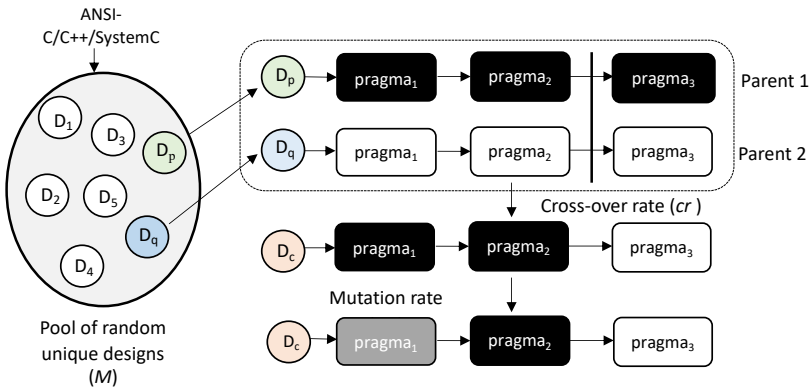



**Figure 6.9:** Genetic Algorithm (GA) HLS DSE overview.

### 6.3.3 Genetic Algorithm

The genetic algorithm (GA) approach is similar to the SA as it relies on the random generation of pragma combinations and then randomly modifies some of these pragmas while it minimizes the cost function, similar to the SA case.

In GA, a pragma is considered as a gene and entire pragma list a chromosome. Two chromosomes are then combined and mutated based on pre-defined crossover and mutation probabilities ($cr$ and $mr$) to produce an offspring (child). Fig 6.9 shows an example of how the GA algorithm works to build a HLS design space explorer. In particular:

**Parse Behavioral description:** Parse the behavioral description, extract explorable operations (arrays, loops and functions) and build a database with all possible pragmas for each operation.

**Initialize Hyper-parameters:** Initialize the crossover ($cr$) and mutation rate ($mr$). Set the cost function weights ($\alpha$ minimize area or $\beta$ latency).

**Step 1: Generate Parents:** Generate two parents with random chromosomes and synthesizing these in order to obtain their area, performance in order to compute their cost.

**Step 2: Crossover:** Generate an offspring ($D_c$) and based on the cross-over rate determine the pragma lists of the two parents to be merged by randomly choosing a cross-over point as shown in Fig. 6.9. This cross-over rate is typically set to a very high value ( 90-95%). If no cross-over is selected then randomly select one of the parents.

**Step 3: Mutation:** Update some of the pragmas based on the mutation rate (typically set to a low value 10-20%). Synthesize the new offspring and compute its cost.

**Step 4: Substitute Parent:** The explorer then continues by comparing the cost of the child vs. its parents and substitutes the parents with highest cost. If the offspring has a higher cost than any of the parents, then it is discarded. Steps 2 and 4 are repeated until no child can improve the cost function for $N$ iterations, where $N$ is a GA specific hyper-parameter. A new pair of parents is generated after this or the exploration loop exited to step 5.s

**Step 5: Cost function update:** Update the cost function weights ($\alpha$ and $\beta$) to explore a different part of the trade-off curve and repeat steps 1 to 4.

**Pareto-optimal Design Extraction:** Once all the combinations have been synthesized, the Pareto-optimal designs are extracted by comparing the area and performance of all of the designs.

All these meta-heuristics are extremely sensitive to their hyper-parameters, which have to be carefully tuned to obtain good results. What makes things worse is that these hyper-parameters might need to be adjusted for benchmarks with different characteristics. Thus, it is imperative to have an automatic method to find these parameters in order to facilitate the use of these meta-heuristics. In [68] we showed how to use machine learning techniques to set these automatically.

### 6.3.4 HLS DSE methods Comparison

Table 6.1 compares all three approaches just described, highlighting their advantages and disadvantages. One of the main advantages of the exhaustive enumeration approach is that it guarantees to find the optimal solution. This is the real Pareto-optimal designs. Unfortunately, for larger benchmarks the search space is too large to complete evaluate all possible combinations. In this case, the only option is to use a heuristic.

SA and GAs have similar trade-offs. They are both easy to implement and have shown to lead to good results. The main problem is that we cannot guarantee that they will produce the optimal results. Thus, we cannot consider the results as Pareto-optimal and call them dominating designs. The question then is how to know if the results are good or bad ? The only way to measure the quality of their results is to run them with different seeds and merge all of the results or even running multiple meta-heuristics and then merge all of the results in order to obtain a reference curve which serves as the bases to measure the quality of the heuristic. One additional weak point of these meta-heuristics is that they are very sensitive to their hyper-parameter settings.

**Table 6.1:** Trade-off simple HLS DSE methods

| Method | Trade-off |
|---|---|
| Exhaustive | + Very easy to implement<br>+ Leads to the optimal solution<br>- Long runtime<br>- Not usable for larger search spaces |
| SA | + Easy to implement<br>+ Leads to good results<br>- Cannot guarantee optimal results<br>- Very sensitive to hyper-parameters |
| GA | + Easy to implement<br>+ Leads to good results<br>- Cannot guarantee optimal results<br>- Very sensitive to hyper-parameters |

It is therefore important that you understand these trade-offs when building your own explorer. One simple and effective approach could be e.g., to first calculate the total number of combinations required to fully explore the search space of a new unseen behavioral description and based on the search space size either perform an exhaustive enumeration or call the heuristic. This would obviously require implementing both approaches.

One obvious way to accelerate the DSE process is by evaluating multiple pragma combinations at the same time by parallelizing the explorer. If the explorer is written in C/C++ this can be done using **pthreads**. The main problem with this approach is that every time that a new pragma combination is generated the explorer requires to invoke the HLS process in order to evaluate the effect of these synthesis options on the resultant design. This tool invocation requires to check out a HLS tool license that will not be released until the HLS process has finished. This implies that the maximum number of parallel threads is limited by the number of licenses available. In the ASIC case, these licenses are extremely expensive, making it often prohibitory for some companies to have more than one. On contrary FPGA vendors provide their HLS tools free. Thus, in [69] we investigate the use of FPGA HLS tools to find the ASIC Pareto-optimal designs.

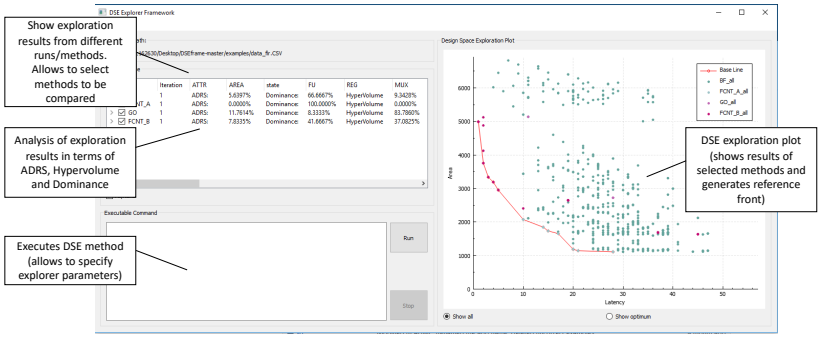



**Figure 6.10:** GUI-based Design Space Exploration framework. Available at [70].

### 6.3.5 Design Space Exploration Framework

To aid anyone interested in creating their own explorer, we have developed a design space exploration framework that works with any heuristic. This framework is Graphical User Interface (GUI) based and allows to call any explorer from within the framework itself. It also displays the exploration results and measures the quality of results of different exploration runs or different methods by computing the different quality metrics introduced in section II E. A screenshot of the framework is shown in Fig. 6.10. The source code and binary have been made public at [70].

The framework is Qt-based. This makes it flexible enough to run on Linux and Windows. It allows anyone creating their own DSE method to display and analyze their results. A documented interface between the explorer and the GUI allows the framework to display the exploration results in the trade-off curve viewer. Multiple runs and multiple explorers can be displayed on the same graph and different quality metrics reported (ADRS, hypervolume and dominance) by automatically creating the reference front. By default it will generate the reference front by combining the best results of all the methods selected.

---

**Example:** Let's look at an example of how prepare a design to be explored easily. Listing 6.3 shows a simple behavioral description that computes the moving average of 8 numbers that we have used throughout the book. It is composed of three *explorable* operations: (1) The array that holds the

8 values, (2) the for-loop that shifts the data in the array to make room for the new value, and (3) the array that adds up all of the values stored in the array. The pragmas that control how these *explorable* operations are synthesized are declared in a separate header files called pragma.h shown in listing 6.2. The way #define statement in the pragma.h header file substitute the actual pragma in the behavioral description to be synthesized. There are obviously different ways to specify the synthesis directives as these are tool dependent. Some HLS tools require you to specify the pragmas in an external tcl script, and hence do not require this header file.

Finally, listing 6.1 shows a library generated by us that include all possible pragma settings for each of the *explorable* operations.

**Listing 6.1**: pragma.lib

```
pragma1 array REG
pragma1 array RAM

pragma2 unroll_times 0
pragma2 unroll_times all
pragma2 unroll_times 2
pragma2 unroll_times 4

pragma3 unroll_times 0
pragma3 unroll_times all
pragma3 unroll_times 2
pragma4 unroll_times 4
```

**Listing 6.2**: pragma.h

```

#define pragma1 array=REG
#define pragma2 loop=all
#define pragma3 loop=all
```

**Listing 6.3**: Behavioral Description for HLS

```
#include "pragma.h"
int buffer[8] = {0, 0, 0, 0, 0, 0, 0, 0}; // pragma1

int ave8(int in_new){
    int  sum,  i;

    /* pragma2 */
    for (i = 7; i > 0; i--)
        buffer[i] = buffer[i- 1];

    buffer[0] = in_new;

    /* pragma3 */
    for (i= 0; i< 8; i++)
        sum += buffer[i];

    return(sum/8);
}
```

---

With this information in hand, it is very straight forward to build an automated explorer. The simplest one would do an exhaustive enumeration of all possible pragma combinations. Although time consuming,

this approach would lead to the optimal Pareto-optimal designs as all possible combinations are evaluated. The exhaustive enumeration would perform the following steps:

**Read Database of pragmas** Read pragma.lib of all possible pragmas for each explorable operations.

**Step 1: Generate a unique pragma combination** Generate a unique pragma combination by going through all the pragmas just read into.

**Step 2: Synthesize (HLS) new pragma combination** Generate a new pragma.h header file that contains the new uniquely generated pragma combination and synthesize it (HLS). It should not be forgotten that this new design has to be fully synthesized including parsing the description (in case that the parsing stage is different from the synthesis stage). Step1 and step 2 are repeated until all possible pragma combinations are generated.

**Pareto-optimal Design Extraction** Once all the combinations have been synthesized, the Pareto-optimal designs are extracted by comparing the area and performance of all of the designs.

## 6.4 Conclusions

In this chapter I have reviewed the main body of work done in the area of HLS design space exploration presented in the recent years and in particular when the HLS process is considered as a black box. Because of the large exploration space, many different dedicated heuristics have been proposed. I have classified these into synthesis-based and model-based. Each method has its advantages and disadvantages that we have discussed in detail.

I have also introduced an open-source Qt-based exploration framework that we have made publicly available to facilitate the creation of HLS design space explorers. This framework also automates the analytical comparison of different techniques. Finally, we listed current DSE method limitations and proposed un-addressed challenges and opportunities.

Summary

- One significant advantage of HLS is that it allows to easily generate functional equivalent designs by only changing the synthesis options.
- HLS has three main tuning knobs: FU constraints, synthesis directives (pragmas) and global synthesis options.
- The objective of HLS DSE is to find the Pareto-optimal designs quickly.
- Meta-heuristics like Simulated Annealing and Genetic Algorithm have been shown to lead to good results for HLS DSE.
- A HLS DSE framework and example of how to explore a simple description has been introduced.

## 6.5 Problems

**HLS DSE**
Implement different HLS DSE methods including (1) the exhaustive search HLS DSE explorer (2) Simulated Annealing based and (3) Genetic Algorithm and compare their runtime vs. quality of results. You can use hypervolume or ADRS to compare the quality of the results. For this, use the CHstone or S2CBench benchmarks and the DSE framework from [70].

# 7 Input Languages

## 7.1 Introduction

Different HLS tools support different languages. Most do support ANSI-C, while others only support SystemC. Most vendors also include their own hardware extensions on top of these common languages to give users more controllability over the generated circuit.

This chapter deals with how HLS vendors support these different input languages and the main differences between these languages.

## 7.2 HLS Tools Language Parsers

Commercial HLS tools are often designed like many SW tools in a modular way where every different part of the HLS tool is a separate binary. The HLS IDE basically calls the specific executable when selected.

Figure 7.1 shows an overview of a typical HLS tool structure with all the possible different individual executable programs ranging from the parsers, to the main synthesis engine (hls.exe) to the different RTL back-ends to either generate Verilog or VHDL (verilogout.exe and vhdlout.exe).

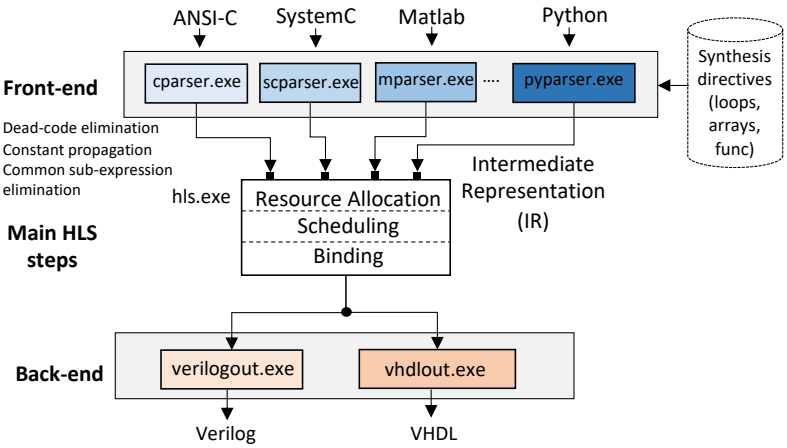



**Figure 7.1:** Modular HLS tool structure overview.

Because HLS tools often accept multiple different input language, they might include different parsers. In the figure shown, one parser (cparser.exe) parses pure ANSI-C code, another SystemC (scparser.exe), another MATLAB code (mparse.exe) and the last Python (pyparser.exe). The output of the parser is an intermediate file representation (IR) that is common to all the parsers. This allows the HLS vendors to easily support new languages by simply adding a new parsers and not needing to modify anything else in the HLS tool flow as the main synthesis engine takes this IR as input.

As we have learned in Chapter 3 the parser is extremely important because it is responsible apart from doing the syntactical checks, for

performing technology independent optimizations. Not doing these efficiently will inevitably lead to larger, less efficient hardware circuits.

As shown in the figure, the parser also reads in the synthesis directives (pragmas), generating a different CDFG based on this.

It is therefore important to consider this when using HLS as the exact same HLS tool might lead to different results when translating one design from one language to another [71]. Some tools do not follow this structure and base their parsers on llvm [16, 72], which is a collection of modular and reusable compiler and toolchain technologies. This has two significant advantages. First, llvm is a very mature, open source, compiler infrastructure. This makes it very robust and hence, does a very good job with the technology independent optimization as the entire community works on these optimizations. Second, it includes clang which is an LLVM native C/C++ compiler. Thus, if the HLS tool only supports C/C++ and/or SystemC then LLVM is enough and there is no need to have separate parsers, making the design and maintenance of the HLS front-end much easier.

## 7.3 Languages Extensions

Traditional SW languages can be fully synthesized into HW circuits through HLS, but unfortunately will not lead to the most optimized circuit. One reason is for example that they do not allow to customize the bit width of the different variables. Defining a variable as 'int' will lead to 32-bits, but a particular variable might only need 28 bits. Thus, custom data types are needed to generate smaller and lower-power circuits.

Moreover, many HLS vendors add hardware extensions to their supported SW languages to give user more controllability of the final circuit. In particular ASIC HLS vendors. E.g., how can a HLS user specify if some inputs or outputs should be registered or not? A global option could set register all primary inputs or outputs, but how to control only some? For this HW extensions are very useful to allow users to specify physical data types.

The next subsections describe some of the most typical languages extensions.

### 7.3.1 Custom Data Types

Custom data types are extremely important in HLS. As shown in Chapter 3, the area and delay of a hardware circuit depends on it. E.g., the scheduler might be able to reduce the number of clock steps required if the delay of the FUs is smaller, and the delay obviously depends on the bit width of the FUs.

Table 7.1 summarizes the bit width of typical data types used in traditional SW programming languages, classified into integer and floating-point.

Based on this, it is not possible for any HLS user to specify variables of arbitrary bit width, and for this particular reason HLS vendors are forced

| Type | name | Bitwidth |
|---|---|---|
| integer | char | 8 bits |
| | short | 16 bit signed |
| | unsigned short int | 16 bit |
| | int | 32 bit signed |
| | long long | 64 bit signed |
| float | float | 32 bit signed |
| | double | 64 bit signed |
| | long double | 128 bit signed |

**Table 7.1:** Bit widths of typical SW data types.

to provide custom data types. Moreover, because floating-point number representation leads to larger and slower hardware circuits, it is also advisable to convert any floating point variables to fixed point, unless larger dynamic ranges and precisions are needed by the application. Thus, HLS vendors also provide custom floating point number representations as well as fixed point data types.

**SystemC Data types:** One of the main advantages of using SystemC is that it comes with its own custom data types and that have been standardize by the IEEE. Thus, most of the HLS vendors support these custom data types.

The next chapter covers SystemC in depth including its data types. Thus, we will skip the details here.

**Algorithmic C Data type (ac_type):** Mentor (now Siemens EDA) Catapult HLS tool initially did not support SystemC. Thus, they developed their own bit-accurate data type library that they called algorithmic C [73], which is freely available online[1].

1: https://github.com/hlslibs/ac_types

The numerical package provides classes for bit-accurate integer, fixed-point, floating-point and complex numbers. Table 7.2 summarizes the main data types supported by the ac data types and their syntax.

| Name | Description | Syntax |
|---|---|---|
| ac_int.h | integer | ac_int<W,S> |
| ac_fixed.h | fixed-point | ac_fixed<W,I,S,Q,O> |
| ac_float.h | floating-point | ac_float<W,I,E,Q> |
| ac_complex.h | complex | ac_complex<T> |

**Table 7.2:** Algorithmic C data types

with:

W: integer representing width of the type. For ac_float it is width of the mantissa.
S: bool parameter representing sign or unsigned of ac_int or ac_fixed type.
I: integer representing integer width.
Q,O: enumeration parameter for quantization (rounding) and overflow modes.
T: Type for ac_complex. T is restricted to AC Numerical types and C++ integer and floating types.

The ac data types also include an interface data type to model channel FIFOs (ac_channel.h).

In order to use these data types, the user needs include the appropriate header file from table 7.2. All the definitions are included in these header

files and thus, it is not required to link any external library. Simply including the header file suffices.

It is important to fully read the specifications as the package author makes some assumption regarding e.g., how negative numbers are represented (two's complement) and the bitwidth of standard data types (char, signed char, etc.).

Table 7.3 shows the ranges of the integer and fixed-point data types as a function of their parameters when they are declared as signed vs. unsigned.

**Table 7.3:** Algorithmic C numerical ranges

| Type | Description | Range |
|---|---|---|
| ac_int<W,false> | unsigned integer | 0 to $2^W$-1 |
| ac_int<W, true> | signed integer | -$2^W$ − 1 to $2^W$ − 1-1 |
| ac_fixed<W,I, false> | unsigned fixed-point | 0 to (1-$2^{-W}$) $2^I$ |
| ac_fixed<W,I, true> | unsigned fixed-point | (-0.5) $2^I$ to (0.5-$2^{-W}$) $2^I$ |

Let's look at some examples of how specifying different bit-accurate data types affect the actual ranges. Table 7.4 shows some examples including the specific ranges for each of the cases for unsigned and signed cases and for integer and fixed-point data types.

**Table 7.4:** Algorithmic C examples

| Type | Range |
|---|---|
| ac_int<1,false> | 0 to 1 |
| ac_int<1, true> | -1 to 0 |
| ac_int<4, false> | 0 to 15 |
| ac_int<4, true> | -8 to 7 |
| | |
| ac_fixed<4,4,false> | 0 to 15 |
| ac_fixed<4,4, true> | -8 to 7 |
| ac_fixed<4,0, false> | 0 to 15/16 |
| ac_fixed<4,0,true> | -0.5 to 7/16 |

---

**Example**
Determine the ranges for the following cases:
ac_int<8,false>
ac_int<12,true>

**Solution**
ac_int<8,false> is an unsigned integer of 8 bits. Hence, the range is: 0 to $2^8$-1= 0 to 255
ac_int<12,true> is a signed integer of 12 bits. Hence the range is: -$2^{12-1}$ to $2^{11}$-1 = -2,024 to 2,023

---

the fixed-point data type case, also allows to specify the overflow and quantization modes. The quantization mode basically defines what should happen when the numerical value exceeds the maximum value that can be represented with the allocated bits, while the quantization mode determines how the fractional part should behave due to the limited number of bits allocated to it.

```
ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_o_mode 0>
```

Table 7.5 summarizes the quantization modes supported. It should be noted that typically the truncation modes lead to the largest numerical errors, but are also the least costly to implement, with mostly no hardware costs involved. On the other hand, the different rounding modes lead to smaller numeric errors, but incur in different levels of overheads as additional logic is required to implement their behavior.

| Mode | Description |
|---|---|
| AC_TRN | Truncated bits. |
| AC_TRN_ZERO | Truncated towards zero |
| AC_RND | Round bits |
| AC_RND_ZERO | Round towards zero |
| AC_RND_INF | Round towards +/- infinite |
| AC_RND_MIN_INF | Round towards - infinite |
| AC_RND_CONV | Round towards even q multiples |
| AC_RND_CONV_ODD | Round towards odd q multiples |

**Table 7.5:** Algorithmic C quantization modes

Table 7.6 summarizes the different saturation modes. Basically, specifying what should happen if the number is too large to be represented with allocated bits.

| Mode | Description |
|---|---|
| AC_WRAP | Drop bits left to MSB (default, no cost) |
| AC_SAT | Saturates to closest min or max value |
| AC_SAT_ZERO | saturates to 0 |
| AC_SAT_SYM | for unsigned treat as AC_SAT<br>for signed on overflow number is min set to closest. |

**Table 7.6:** Algorithmic C saturation modes

To use the ac data types, the user has to simply include the header file that includes the required data types and then declare variables as shown previously. E.g.,

```
#include <ac_int.h>

ac_int<8, true> p; // p is 8 bit signed.
ac_int<11,false> q; //q is 11 bits unsigned
```

**Note:** It is extremely important to understand how the different data types behave when doing different bitwise and arithmetic operations to it. For this it is important to consult the reference manual and simulate the behavioral description once the data types have been refined to verify the numerical stability of the application with the new data types, comparing the results vs. the original SW description.

Siemens EDA also claims that these data types are faster than SystemC data types.

---

**Example**
Think about how you could test Siemens's claims that the AC data types simulate faster than SystemC's data types.

**Solution**
Write two identical program. One that has different has ac_int data types of different bit widths and one with SystemCs' sc_int data types and perform multiple arithmetic operations in a loop with a larger number

of iterations. Compile both programs and measure the time that it takes each program to finish the simulation. Do the same for ac_fixed and sc_fixed for different bit widths. The programs themselves do not need to do anything useful.

---

**All Purpose Data type (ap_type):** Xilinx released their own bit-wise accurate data types called 'All Purpose (ap)' data types with Vivado HLS. The syntax of this bit-accurate data types is very similar to the AC data types from Siemens.

Table 7.7 gives an overview of the header files needed and syntax. One main difference with the AC data types is that by default the AP data types are signed and to make them unsigned a 'u' needs to be specified in the declaration.

**Table 7.7:** All purpose data types

| Name | Description | Syntax |
|---|---|---|
| ap_int.h | integer | ac_[u]int<W> |
| ap_fixed.h | fixed-point | ap_fixed<W,I,Q,O,N> |

It should be noted that the AP data types are only usable when writing C++ and not ANSI-C. The main reason for this is that C++ allows templates while ANSI-C does not. Thus, making C++ code written with these data types still fully compilable.

Here is an example of how to use the AP data types:

```
#include <ap_int.h>

ap_int<9> p; // p is 9 bit signed.
ap_uint<11> q; //q is 11 bits unsigned
```

In the case of fixed-point data types, the AP data types supports the same overflow and quantization modes than the AC data types. Table 7.8 show the quantization modes and their description.

**Table 7.8:** All purpose quantization modes

| Mode | Description |
|---|---|
| AP_TRN | Truncated bits. |
| AP_TRN_ZERO | Truncated towards zero |
| AP_RND | Round bits |
| AP_RND_ZERO | Round towards zero |
| AP_RND_INF | Round towards +/- infinite |
| AP_RND_MIN_INF | Round towards - infinite |
| AP_RND_CONV | Round towards even q multiples |

Table 7.9 summarizes the different saturation modes. Basically, specifying what should happen if the number is too large to be represented with allocated bits.

**Table 7.9:** Algorithmic C saturation modes

| Mode | Description |
|---|---|
| AP_WRAP | Drop bits left to MSB (default, no cost) |
| AP_WRAP_SM | Sign, magnitude wrap around |
| AP_SAT | Saturates to closest min or max value |
| AP_SAT_ZERO | saturates to 0 |
| AP_SAT_SYM | for unsigned treat as AC_SAT<br>for signed on overflow number is min set to closest. |

Here is an example of how to use the AP fixed-point data types:

```
#include <ap_fixed.h>

ap_fixed<22,8, AP_RND> p; // p is 22 bits, 8 int, 16 fractional
```

### 7.3.2 Hardware Extensions

Some HLS vendors also add extensions to the input languages to make it easier f to build the exact hardware circuit that the user has in mind.

One example is NEC's CyberWorkBench HLS tool which apart from synthesizing ANSI-C and SystemC also takes as input BDL . BDL stands for behavioral description language and it is basically C with hardware extensions.

Table 7.10 summarizes the main features of BDL:

| Extension | Syntax | Description |
|---|---|---|
| Physical type | ter/reg/var/mem | Specify physical variable type |
| Bit width | var(MSB..end) | bitwise data type |
| IO | in/out/inout | IO port declaration |
| top function | process | top module to be synthesized |
| Operators | :: | Concatenation |
| | &>, \|>, | bit reduction operators |
| Timing | $ | clock step descriptor |
| Constanbt | HiZ | high impedance |

**Table 7.10:** Features of BDL language

Listing 7.2 shows an example of a basic BDL program that multiplies the values of two inputs. The main differences between BDL and C is that the IO ports need to be described and the top model specified with the keyword **process**.

Listing 7.2 shows the same example, but in this case the inputs and outputs are defined as register type instead of terminal type which is simply a 'wire'.

```
1 in ter(7..0) a, b;
2 out ter(15..0) c;
3 process mult()
4 {
5 c = a * b;
6 }
```

```
1 in reg(7..0) a, b;
2 out reg(15..0) c;
3 process mult_reg()
4 {
5 c = a * b;
6 }
```

**Listing 7.2: BDL example** registers

It is not possible in ANSI-C to have this amount of controllability which is particularly useful in the ASIC design case, where the hardware needs to follow very precise timing. One of the problems with BDL is that it is not standard ANSI-C code anymore and hence it cannot be compiled with a regular C compiler (e.g., gcc). One way to address this is to have the pure SW version and the BDL version on the same C file and use pre-compiler directives to either compile the SW version or to synthesize the BDL version as shown in listing 7.3.

**Listing 7.3**: C and BDL on same file

```
#ifdef C // for ANSI-C version
   char a, b;
   short c;
   void mult()
#else  // for BDL version
   in ter(7..0) a, b;
   out ter(15..0) c;
   process mult()
#endif
{
c = a * b;
}
```

Compiling the program defining C would generate the purse SW program (**%gcc -DC**), while not defining C would synthesize the BDL version.

## 7.4 Languages

This section reviews the most popular languages supported by current commercial HLS tools and tries to give an understanding and their different trade-offs. It is important for anyone starting to use HLS to decide which language to use and this has some important future implications.

### 7.4.1 ANSI-C

C was developed in the Bell Labs by Dennis Ritchie between 1969 and 1973 while he was working on Unix operating system. He named the language C because there had been two other programming languages named A and B.

The American National Standards Institute (ANSI) formed a committee, to establish a standard specification of C in 1983. The prerelease standard C was published in 1988, and sometimes referred to as C88. The ANSI standard was completed in 1989 (C89). Since then multiple revisions of the standard have been released, e.g., C90, C95, C11 and C17.

C was widely adopted in many domains ranging from embedded systems to the design of operating systems (Unix was written in C). Because of its popularity it has an extremely wide user base making it a perfect candidate to be adopted for HLS.

ANSI-C is supported by the most widely used compilers like gcc and clang.

### 7.4.2 C++

C++ is a high-level programming language created by Bjarne Stroustrup as an extension of the C programming language since 1979. Initially he called the language C with Classes and in 1983 renamed it C++ to signify its advancement over C (++ is the increment operator in C).

C++ was initially standardized in 1998 which was then amended by the C++03, C++11, C++14, and C++17 standards. The current C++20 standard supersedes these with new features and an enlarged standard library. The standard is reviewed every three years.

C++ has some unique features like classes, inheritances, virtual functions and function overloading.

### 7.4.3 SystemC

SystemC is a C++ class that was originally developed to model hardware. Its template-based implementation makes it easy to write C++ code with hardware extensions while making it compilable using any C++ compiler. Some of these extensions include the ability to declare the hardware's module entity (IO ports and their bitwidth).

One additional characteristic of SystemC is that it allows to simulate concurrency that we encounter in every hardware system. This is done through the use of multiple threads. One per hardware module.

SystemC was recently standardize by the IEEE in the 1666 LRM [25]. Accelera also released a synthesizable subset for HLS [26]. Because of this the language has been very well received in the VLSI design community, and hence, all ASIC HLS tools support synthesizing SystemC.

FPGA vendors do not support SystemC as it is considered by many FPGA users as too low-level. The next chapter goes over SystemC in detail.

### 7.4.4 MATLAB

MATLAB stands for *matrix laboratory*. The main difference between MATLAB and other programming languages is that other languages work with numbers one at a time, while MATLAB operates on whole arrays and matrices.

Due to this MATLAB is often used in signal and image processing applications. MATLAB also makes extensive use of add-on toolboxes in the form for APIs that are often commercialized separately.

MATLAB does not require declaring variables, specifying data types, or allocating memory. In many cases, MATLAB eliminates the need for 'for' loops. Because of this, a single line of MATLAB code can often replace several lines of C or C++ code. The language also includes APIs to visualize the results mainly by plotting them as different types of graphs.

The code below shows how to design low-pass FIR filter with a 20Khz cut-off frequency and 96Khz sampling frequency.

```
1 N  = 100;           % FIR filter order
2 Fp  = 20e3;     % 20 kHz passband-edge frequency
3 Fs  = 96e3;     % 96 kHz sampling frequency
4 Rp  = 0.00057565;   % Corresponds to 0.01 dB peak-to-peak ripple
5 Rst = 1e-4;     % Corresponds to 80 dB stopband attenuation
6
```

**Listing 7.4**: MATLAB FIR filter

```
eqnum = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge');
fvtool(eqnum,'Fs',Fs) % Visualize filter
```

This FIR filter example clearly highlights the benefits, but also shortcomings of using MATLAB in the context of HLS. The advantages are clear from the SW perspective. The built-in libraries specifically targeted for specific applications allow to quickly design and optimize complex image and DSP applications.

The main drawback is how to synthesize these APIs? HLS requires to have full access to the code within the API. That is why in the past only MATLAB itself could synthesize MATLAB descriptions through their HDL coder. MATLAB's HDL coder does not perform a traditional HLS on the MATLAB code, but instead it maps the code to a pre-defined template. Basically every SW API has a HW template that is then configured based on the parameters passed to the model.

Recently Cadence announced that it can also synthesize MATLAB code through their commercial HLS tool Stratus [74].

### 7.4.5 OpenCL

OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse hardware platforms, e.g., supercomputers, data centers personal computers, mobile devices and embedded systems. OpenCL was created and is managed by the Khronos Group.

OpenCL is widely used throughout the industry. Many silicon vendors ship OpenCL with their processors, including GPUs, DSPs and FPGAs. The OpenCL API specification enables each chip to have to have its OpenCL drivers tuned to its specific architecture. Having the same, standardized API available on many systems enable developers to widely deploy their applications to reach more customers while minimizing porting and support costs.

The big idea of OpenCL is that it substitutes loops with functions (kernels) executing concurrently. e.g. a 512x512 image with one kernel invocation per pixel implies 512x512= 262,144 kernel executions.

The listings below show an example of a program doing a matrix multiplication in C (listing 7.5) and in OpenCL (listing 7.6)

**Listing 7.5**: Matrix multiplication in C

```
void mul(   int n,
        float*a,
        float*b,
        float *c){
  int i;
  for(i=0; i < n; i++)
    c[i] = a[i]*b[i];
}
```

**Listing 7.6**: Matrix multiplication in OpenCL

```
__kernel void mul(
    __global float *a,
    __global float *b
    __global float *c) {
  int d = get_global_id(0);
  c[id] = a[id] * b[id];
  }
```

One of the characteristics of OpenCL is that it assumes a CPU+accelerator structure.

With regard of using OpenCL to build custom hardware accelerators with HLS, only Intel OpenCL SDK currently can synthesize OpenCL

descriptions onto their FPGAs. Similar to the MATLAB HDL coder, the OpenCL SDK maps any OpenCL description to a template architecture and thus, does not perform a traditional HLS. This approach is more suitable for SW designers with no prior HW knowledge that want to accelerate a specific application on an FPGA board.

### 7.4.6 Languages Comparison

Table 7.11 summarizes the pros and cons of the different languages reviewed in this subsection. It is important to understand the benefits and limitations of each of them and decide accordingly based on the individual needs and background.

| Languages | Trade-off |
|---|---|
| ANSI-C | + Large user base<br>+Simplicity<br>- Cannot model concurrency<br>- Lack of flexibility (no templates) |
| C++ | +Templates gives flexibility (data types)<br>-Object oriented programming complexity |
| SystemC | +Built-in bit-accurate data types<br>+Can model concurrency<br>+Standardize by IEEE<br>-Object oriented programming complexity<br>-More lines of code to write |
| MATLAB | +Very good simulation environment<br>- Not supported by most HLS tools |
| OpenCL | +Very portable<br>- Limited HLS tool support<br>-Quality of HLS result |

**Table 7.11:** HLS supported languages trade-offs

## 7.5 Summary

In this chapter we have reviewed the different input languages supported by most HLS tools.

**Summary**

- Most HLS vendors accept C, C++ and SystemC.
- LLVM is often used as front-end or dedicated parsers.
- The quality of the synthesized circuit will also depend on the quality of the parser that does technology independent optimizations.
- HLS vendors also often have their own HW extensions to generate better circuits.

# SystemC 8

## 8.1 Introduction: What is SystemC and Why?

SystemC is a C++ class for hardware design. It was original designed to allow the generation of fast hardware simulation models. Now almost all HLS tools synthesize SystemC descriptions into RTL since OSCI (now Accelerate) published a synthesizable SystemC subset [26], at which I participated in.

One of the reasons for its popularity is that the IEEE standardized SystemC in the Language Reference Manual (LRM) 1666 in 2012 [25], including SystemC's data types which allow to specify any arbitrary bit width including fixed-point data types. As we have shown this is extremely important in hardware design as it significantly affects the circuit area and delay.

The question that we might ask ourselves is what is special about SystemC and why was there a need for it. Here is a list of the most important reasons:

1. SystemC allows to model concurrency. C/C++ has no notion of parallelism, but hardware is executed in parallel (multiple HW modules operated in parallel).
2. C/C++ have no notion of time/clocks.
3. C/C++ do not provide custom data types.
4. Hardware requires specific communication interfaces like signals and FIFOs.



## 8.2 How to use SystemC?

Before we start writing SystemC code we have to install it in our computer. SystemC is freely available from www.accelera.org. Follow these steps:

1. Visit www.accelera.org and download the latest SystemC version (e.g., systemc-2.2.0.tgz).
2. Unzip the SystemC package (%tar -xvf systemc-2.2.0.tgz) and follow the installation instruction given in the README file. This should generate a libsystemc-2.3.0.so static library that needs to be linked with every new SystemC code written.
3. Write your own SystemC file including #include "systemc.h" header file.

I created different YouTube videos showing the installation processing[1] SystemC can be compiled using a standard C++ compiler (e.g., g++) into an executable file that can then be executed as shown in Figure 8.1 and as follows:

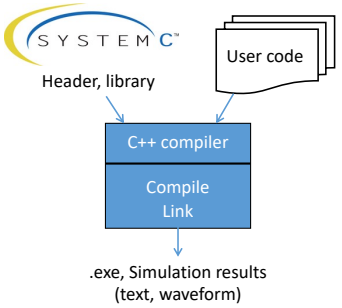



**Figure 8.1:** Overview of how to compile SystemC program.

1: https://www.youtube.com/user/DARClabify

```
%g++ -o test.exe test.cpp -I$SYSTEMC_HOME/include -
L$SYSTEMC_HOME/lib-linux -lsystemc
```

This will generate test.exe that can be fully executed.

You might want to specify the path where the SystemC package was compiled as an environment variable in LINUX so that you do not have to specify the full path every time you want to compile a SystemC file.

## 8.3 Writing SystemC Programs

SystemC makes use of C++ macros to hide verbose C++ syntax and to provide information to the SystemC kernel (scheduler), which is responsible of execute the multiple threads in each SystemC program. These macros, e.g., SC_MODULE , SC_CTOR are not SystemC keywords, but C++ macros. This is one of the main reason why g++ can be used to compile SystemC programs.

The most common way to write SystemC programs is to separate the *entity* from the *functional description*. The entity includes the IOs and describes other issues like the clock type, reset and type of hardware circuit (combinational or sequential). The functional description contains the main function of the circuit. Basically SystemC is a wrapper around an ANSI-C description where the functional description is the same.

**IOs:** Primary inputs and outputs (IOs).
**Clock:** Clock name and type (rising of falling edge)'
**Reset:** Rest name and type (asynchronous or synchronous, active high or low).
**Sensitivity:** Sensitivity list of inputs that cause outputs to change.
**Circuit type:** Type of hardware circuit: Combinational or sequential.

The functional description portion of the SystemC program can be defined in three different ways:

**sc_method:** Program is executed when sensitivity list is triggered. The process runs to completion before returning control to the SystemC scheduler. Cannot have any wait() statements. Mostly used to model combinational circuits. Cannot save variables or control state between invocations. **Synthesizable**.
**sc_cthread:** Models clocked digital logic. Process is only called once when program is started. Execution is suspended with wait() statement, and relinquish control back to SystemC scheduler that then executes another thread. Sensitive to clock edges. May save variables and control state between invocations. **Synthesizable**.
**sc_thread:** Similar to sc_thread, but does not require a clock. Designed to model anything. Sensitive to changes in sensitivity list. Thus it is **non-synthesizable**.

SystemC, contrary to C/C++, allows to specify the name and activation edge in the behavioral description as shown in table 8.1.

---

**Example:** Let's look at an example of how to model a D-flip flop in SystmC using sc_method and sc_cthreads:

| Syntax | Description |
|---|---|
| sensitivity << clk.pos(); | Rising edge of clock |
| sensitivity <<clk.neg(); | Falling edge of clock |

**Table 8.1:** SystemC clock declaration

**Listing 8.2**: D-flip-flop using sc_cthread

```
#include "systemc.h"
SC_MODULE(dff) {
  sc_in<bool > clk;
  sc_in<bool > din;
  sc_out<bool > dout;

  void func() {
   while(true) { // forever
    wait();
    dout.write(din.read());
    }
   SC_CTOR(dff) {
   SC_CTHREAD(func); //once
   sensitive << clk.pos();
  }
};
```

**Listing 8.1**: D-flip-flop using sc_method

```
#include "systemc.h"
SC_MODULE(dff) {
  sc_in<bool > clk;
  sc_in<bool > din;
  sc_out<bool > dout;

  void func() {
  dout.write(din.read());
  }
  SC_CTOR(dff) {
    SC_METHOD(func);
    sensitive << clk.pos();
  }
};
```

In the example we can observe the similitudes and also the main differences between sc_methods and sc_cthreads. The entity is in both cases exactly the same.

The method case is executed every time that the sensitivity list changes, in this case the clk, while in the sc_cthread is modeled as a C++ thread that is only executed once. Because of this. The sc_cthread needs to be executed in an infinite while loop (while(true)) as SystemC will only execute this thread once. Within this while loop, a wait() statement is needed to manually allow the thread scheduler to execute any other thread in the program. Basically, the designer needs to place these wait() signals strategically within the different threads such that the SystemC program can successfully switch between threads.

**Example**

Write the SystemC code, entity (.h) and functionality (.cpp) to model the full-adder shown in Figure 8.5 and the g++ command line to compile it into an executable program.

**Solution**

The following code listing (listings 8.3 and 8.4) show the code of the combinational full-adder written in SystemC. Listing 8.3 shows the entity, while 8.4 specifies the actual functionality of the adder computing the sum and the carry out.

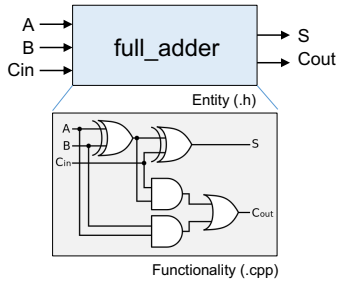



**Figure 8.2:** Overview of full-adder separated into entity (.h) and functional description (.cpp).

**Listing 8.3**: fulladder.h

```
// full_adder.h
#include <systemc.h>

SC_MODULE(full_adder) {
  sc_in<bool> A, B, Cin;
  sc_out<bool> S, Cout;

```

```
  void proc_full_adder();
  SC_CTOR(full_adder) {
    SC_METHOD (proc_full_adder);
    sensitive << A << B << Cin;
  }
};   // do not forget ;
```

**Listing 8.4**: fulladder.sc

```
// full_adder.cpp
void full_adder::proc_full_adder() {
  S = A ^ B ^ Cin;
  Cout= (A & B) | ( Cin & (A ^ B));
}
```

This SystemC program can be compiled by calling g++ as follows:

```
%g++ -o full_adder.exe full_adder.sc -I$SYSTEMC_HOME/include -I.
-L$SYSTEMC_HOME/lib-linux -lsystemc
```

---

### 8.3.1 Testbenches

The main problem with compiling the SystemC programs just illustrated is that the user will not be able to see anything. For this we need a testbench that drives the inputs of the SystemC module and observe the generated outputs. Fortunately, SystemC allows to easily generate testbenches.

For this SystemC requires a main function called sc_main . Each SystemC program can only have 1 sc_main, which is not a module but a function used to instantiate the top level. The sc_main instantiates all of the modules, connects them together and drives any input stimuli. SystemC also allows to trace any signals and dump them into a VCD file that can then be seen with any waveform viewer like gtkwave (open-source waveform viewer).

Listing 8.5 shows an example of how to instantiate the 1-bit full adder as a DUT in the sc_main and how to drive the three inputs by setting different input combinations. When compiled and executed, this SystemC program will produce a VCD file called fulladder.VCD that shows the input and outputs of the simulated full adder.

**Listing 8.5**: sc_main full_adder example

```
#include "systemc.h"
#include "full_adder.cpp"

int sc_main (int argc, char*
    argv[]) {
   sc_signal<bool>   A, B, Cin;
   sc_signal<bool>   S, Cout;

   // Connect the DUT
    full_adder adder("
    fulladder1bit");
   adder.A(A);
   adder.B.(B);
   adder.Cin(Cin);
   adder.S(S);
   adder.Cout(Cout);

  // Create VCD file
  sc_trace_file *wf =
    sc_create_vcd_trace_file("
    fulladder");
  // Dump the desired signals
  sc_trace(wf, A, "A");
  sc_trace(wf, B, "B");
  sc_trace(wf, Cin, "Cin");
   sc_trace(wf, S, "S");
  sc_trace(wf, Cout, "Cout");

```

```
25    A=0;B=0; Cin=0;
26    sc_start(1);
27    A=0;B=0; Cin=1;
28    sc_start(1);
29    A=0;B=1; Cin=0;
30    sc_start(1);
31    A=0;B=1; Cin=1;
32    sc_start(1);
33    A=1;B=1; Cin=1;
34    sc_start(1);
35
36    cout << "@" << sc_time_stamp()
          <<" Terminating simulation
        \n" << endl;
37    sc_close_vcd_trace_file(wf);
38    return 0;// End simulation
39  }
```

## 8.4 SystemC Syntax

The next subsections describe in detail the SystemC syntax, starting with the built in data types, followed by the macros used to read and write from the IOs and how to model real numbers through fixed-point data types.

### 8.4.1 SystemC Data Types

One of the main strengths of SystemC is that it has its own hardware-centric data types. These allow to model any data type that you can encounter in a traditional HDL and also additional ones like fixed-point data types. Because these data types are mainly declared as templates in C++, they can be compiled by any C++ program and hence, accuracy analysis e.g., when doing data type quantization from floating-point to fixed-point data types can be easily and quickly done at the software level.

Table 8.2 summarizes the different IO port data type declarations that SystemC has, including bidirectional ports that are common in hardware circuits.

Table 8.3 shows the most common SystemC data types.

SystemC has also other data types to convey additional information like time when generating testbenches.

```
sc_time clk(10,SC_NS)
```

where the time unit can be SC_PS, SC_NS, SC_MS, etc.

Knowing the IO types and different data types allows us to specify any entity of any bit width. E.g.

```
sc_in<sc_logic> inport;
sc_out<sc_uint<8> > outport;
sc_input<sc_lv<18> > bidirport;
```

| Port type | Description |
|---|---|
| sc_in | Inputs |
| sc_out | Outputs |
| sc_inout | Bi-directional port |

**Table 8.2:** SystemC IO types

**Table 8.3:** SystemC Data types

| Data type | Description |
|---|---|
| sc_bit | 1-bit, uses native C++ bool type (fast) |
| sc_bv | Bit-vector, faster in simulation than sc_lv |
| sc_logic | 1-bit, only supports X,Z,0, 1 |
| sc_lv | Bit-vector, only supports X,Z,0, 1 |
| sc_int | signed integer from 1 to 64-bits |
| sc_uint | unsigned integer from 1 to 64-bits |
| sc_bigint | Arbitrary size signed integer (slow) |
| sc_biguint | Arbitrary size unsigned integer (slow) |
| sc_fixed | Templated signed fixed-point |
| sc_ufixed | Templated unsigned fixed-point |
| sc_fix | Untemplate fixed fixed-point |
| sc_ufix | Untemplated unsigned fixed-point |

**Table 8.4:** SystemC methods to read and write from IOs

| Syntax | Description |
|---|---|
| inport = 1 | Not recommended |
| inport.read() | Recommended |
| outport = 0xFF | Not recommended |
| outport.write(0xFF) | Recommended |

Note the added space between the two closing angled brackets (<>).

### 8.4.2 Reading and Writing from Ports

It is recommended to use port methods to read and wrote from the IOs instead of simply assigning values to the ports. For this SystemC provides two method: read() and write() as shown in table 8.4

### 8.4.3 Signals

SystemC provides signals to convey information between or within modules. These signals are directionless and are basically wires between different parts of the modules. These signals can be of different bitwdith and type as shown in the example below:

```
sc_signal<sc_uint<16> > sig1, sig2;
sc_signal<bool> reset;
```

### 8.4.4 Real Numbers

A real number can be represented as a floating-point data type or a fixed-point data type. Fixed-point data types are most of the time preferred in hardware design because they lead to smaller and faster hardware implementations. Table 8.5 gives an overview of the advantages of using fixed-point data types vs. floating-points.

Because of the advantages of fixed-point data types when generating hardware circuits, SystemC includes a library of fixed-point data types:

```
sc_fixed<wl, iwl, q_mode, o_node, nbits > name(initial);
```

| Data Type | When to use |
|---|---|
| Fixed-point | Smaller/cheaper hardware circuit<br>Fast hardware implementation |
| Floating-point | Larger dynamic range<br>Greater precision (gap between adjacent numbers) |

**Table 8.5:** Comparison between floating-point and fixed-point data types

| Mode | Description | |
|---|---|---|
| Quantization | SC_RND | Round |
| | SC_RND_ZERO | Round towards zero |
| | SC_RND_MIN_INF | Round towards infinity |
| | SC_TRN | Truncate |
| | SC_TRN_ZERO | Truncate towards zero |
| Overflow | SC_SAT | Saturate |
| | SC_SAT_ZERO | Saturate to zero |
| | SC_WRAP | Wrap around |
| | SC_WRAP_SM | Wrap around keeping signed bit |

**Table 8.6:** Supported quantization and overflow modes

---

**Example:** Declare a signed fixed-point signal called 'data' that has a total bitwidth of 8-bits dedicating 2-bits to the integer part and the rest to the fractional part and initialize the signal to 1.75.

**Solution:** sc_fixed<8, 2 > data(1.75); $1.75_{10} = 01.110000_2$.

---

**Note:** Before using the SystemC fixed-point data types, you need to enable the use of fixed-point data types before anything else in your code including:

```
#define SC_INCLUDE_FX
```

One of the main problems when using fixed-point data types is how to specify the behavior when your result doesn't land exactly on a representable number or if the number is too large to be represented with the allocated bit width.

---

**Example:** Declare a signed fixed-point signal called 'data' that has a total bitwidth of 10-bits dedicating 8-bits to the integer part and the rest to the fractional part and initialize the signal to -10.25 with quantization mode set to truncation and saturation mode to saturation.

**Solution:** sc_fixed<10, 8, SC_TRN, SC_SAT > data(-10.25);

$10.25_{10} = 00001010.01_2$ two's complement = -10.25 = 11110101.11

---

Figure 8.3 shows visually the effect of the different quantization modes on the numerical output.

Figure 8.4 shows visually the effect of the different saturation modes on the numerical output.

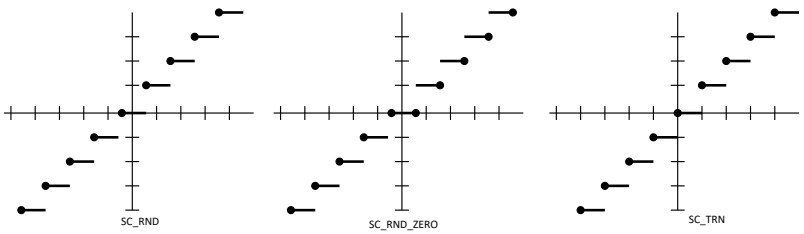



**Figure 8.3:** Overview of different quantization modes.

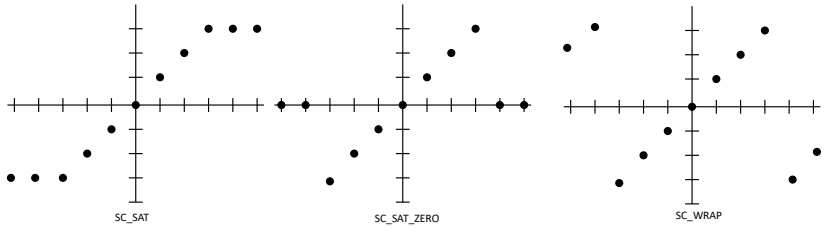



**Figure 8.4:** Overview of different saturation modes.

### 8.4.5 Data type Operations

As any HDL, SystemC allows the selection of partial bits and the concatenations of different signals as shown in table 8.7

---

**Example:** Read an 8-bit variable called 'indata', and rotate it left by one bit and write the output to another 8-bit variable called 'odata'.

**Solution:**
odata.write( (indata.read()(6,1) indata.read()(7)) ); or
odata.write( (indata.read().range(6,1) indata.read()(7)) );

---

### 8.4.6 Logical and Equality Operations

Fortunately, SystemC is still C++, and thus, the logical and equality assignments are still the same as in C/C++ as shown in table 8.8.

**Table 8.7:** SystemC methods to read and write from IOs

| Operation | Example | Description |
|---|---|---|
| Bit select | data(3) | Extracts bit 4 from ''data |
| Partial extract | data(6,3) | Extract bits 6 down to 3 |
| Alt. Partial extract | data.range(6,3) | Extract bits 6 down to 3 |
| Concatenate | (data(0),data(3,1)) | rotates bits |
| Alt. Concatenate | .concat(data(0),data(3,1)) | rotates bits |

| Operation | Description |
|---|---|
| $ \| ∧ ~ | bitwise AND, OR, XOR, NOT |
| >><< | Shift right and left |
| = | Signal assignment |
| == != | Equality check |
| < <= > >= | Comparison |

**Table 8.8:** SystemC logical and equality operations

## 8.5 Modelling Concurrency

One of the main differentiating factor between C/C++ and SystemC is that SystemC has been designed in such a way that it allows to model concurrent module executions. This is basically done by the SystemC scheduler, which is part of the compiled library that is linked into the SystemC program.

Hardware is intrinsically concurrent, meaning that multiple modules operate in parallel. Standard programming languages cannot capture this concurrency. In SystemC this is possible through multi-threading, where every module has its own thread. The key question when modeling large SystemC programs consisting of multiple threads is when and how to switch between the different threads?

SystemC has a central scheduler that controls how the threads are executed. The wait() statements entered by the user in each thread returns the control to this central scheduler, which sets the order in which the threads will be executed. The SystemC scheduler will call each process during the start of the simulation (at time 0) to initialize each process. Users can use 'dont_initialize();' in the SC_CTOR to avoid this.

A very simple example is shown in Figure 8.5 where two modules specified as separate sc_cthreads are modeled together. Module1 has one sc_cthread, while module2 has two sc_threads. The first one sends data to Module1 while the second thread reads the value returned from module1. Through couts placed in each thread we can easily observe how the SystemC scheduler is jumping from one thread to another. Try playing with the wait() statements in the different threads and see how this affects the execution of the program.

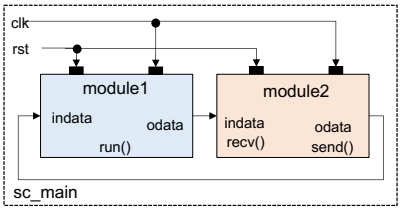



**Figure 8.5:** Overview of system with two modules, module1 having 1 sc_cthread(run()), while module2 consisting of 2 sc_cthreads (recv() and send()).

Listing 8.6 shows the top module with the sc_main for the system shown in Figure 8.5. The next listings describe each of the modules (module1.h/.cpp and module2.h/.cpp).

**Note:** The indata signal of module1 is connected to the odata signal of module2 and vice versa as the input of one module is the output of the other.

```
1  #include "module1.h"
2  #include "module2.h"
3
4  int sc_main(int argc, char**
       argv){
5    sc_clock  clk("clk", 25, SC_NS
       , 0.5, 12.5, SC_NS, true);
6    sc_signal<bool>   rst;
7
8    sc_signal<sc_uint<8> > indata;
9    sc_signal<sc_uint<8> >odata;
10
11   module1 u_module1("module1");
12   module2 u_module2("module2");
13
14   //connect to module1
15   u_module1.clk( clk );
16   u_module2.rst( rst );
```

**Listing 8.6**: sc_main of two modules

```
  u_module1.indata( indata);
  u_module1.odata(  odata );

  // connect to module2
  u_module2.clk( clk );
  u_module2.rst( rst );
  u_module2.indata( indata );
  u_module2.odata( odata );

 // Start simulation
 sc_start( 25, SC_NS );
 rst.write(0);
 sc_start( 25, SC_NS );
 rst.write(1);
 sc_start();
  return 0;
};
```

**Listing 8.7**: module1.h

```
#ifndef MODULE1_H
#define MODULE1_H
#include "systemc.h"

SC_MODULE(module1)
{
    sc_in<bool> clk;
    sc_in<bool> rst;
   sc_in<sc_uint<8> > indata;

   sc_out<sc_uint<8> > odata;

   void run() ;

   SC_CTOR(basic)
     {
       SC_CTHREAD(run, clk.pos()
    ) ;
       reset_signal_is(rst,
    false) ;
     }

   ~module1(){}
};
#endif // MODULE1_H
```

**Listing 8.8**: module1.cpp

```
#include "module1.h"

void module1::run(){
  sc_uint<8>  data_in, data_out;
  int x=0;

  // RESET state
  data_in =0;
  data_out=0;
  cout << endl << "Reset[" << x
    << "]=" << data_out;
  wait();

  while(1){
    // READ input data
    data_in = indata.read() ;

    // UPDATA data read
    data_out = data_in + 10;

    cout << endl << "M1 Data["
    << x << "]=" << data_out;
    x++;

    // WRITE data back
    odata.write(data_out);
    wait();
  }
}
```

**Listing 8.9**: module2.h

```
#ifndef MODULE2_H
#define MODULE2_H
#include "systemc.h"

SC_MODULE(module2)
{
    sc_in<bool> clk;
    sc_in<bool> rst;
   sc_in<sc_uint<8> > indata;

   sc_out<sc_uint<8> > odata;

  void recv();
  void send();

   SC_CTOR(module2)
     {
    SC_CTHREAD(send,clk.pos());
    reset_signal_is(rst,false);

    SC_CTHREAD(recv,clk.pos());
    reset_signal_is(rst,false);
     }

   ~module2(){}
};
#endif // MODULE2_H
```

```cpp
#include "module2.h"

// Send data thread
void module2::send(){
  int x=0;
  unsigned  in_read;

  // Reset state
  odata.write(0);
  cout << endl << "Sending Reset
    [" << x << "]=" << x;
  wait();

  while(true){
    wait();

    cout << endl << "Sending
    data[" << x << "]=" << x;
    odata.write(x);

    if(x == 5) // only send 5
    data
      sc_stop();

    x++;
  }//while_loop
}


// Receive data thread
void module2::recv(){
  unsigned int out_write=0;
  int x=0;

  cout << endl << "Receiving
    Reset[" << x << "]=" << x;
  wait();

  while(true){
  wait();

    out_write = indata.read();
    cout << endl << "Receiving
    data[" << x << "]=" <<
    out_write;
    x++;
  }
}
```

**Listing 8.10**: module2.cpp

**Example:** Remove the wait() statement from the main while-loop in module1. What happens? Why?

## 8.6 Hierarchical SystemC

Similarly to any HDL, SystemC allows to instantiate multiple modules in the same system. This allows to create structural SystemC program as well as behavioral systems, similarly to Verilog or VHDL.

Figure 8.6 shows an example of a 4-bit shift register that needs to be modeled in SystemC. In this case we assume that the D-FF has already been modeled as shown before and that we want to build the 4-bit counter by instantiating four of these D-FFs.

Listing 8.11 shows how this is accomplished through the constructor, instantiating four D-FFs each initialized with a different name.

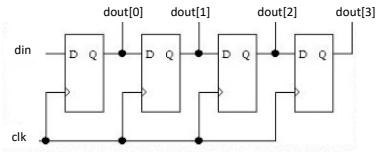



**Figure 8.6:** Structure of 4-bit shift register

```cpp
// shiftreg_struct.h
#include <systemc.h>
#include "dff.h"

SC_MODULE(shiftreg_struct) {
    sc_in<bool> clk;
    sc_in<bool> din;
    sc_out<sc_uint<4> > dout;

```

**Listing 8.11**: shiftreg_struct.h

```
    sc_signal<bool> q1, q2, q3, q4;

    // D-FF instantiations
    dff dff1, dff2, dff3, dff4;

    //Constructor
    SC_CTOR( shiftregstruct ) : dff1 ("dff1"), dff2("dff2"), dff3(
     "dff3"), dff4("dff4"){
    dff1.clk(clk);
    dff1.din(din);
    dff1.dout(q1);

    dff2.clk(clk);
    dff2.din(q1);
    dff2.dout(q2);

    dff3.clk(clk);
    dff3.din(q2);
    dff3.dout(q3);

    dff4.clk(clk);
    dff4.din(q3);
    dff4.dout(dout);
    }
};
```

Where in this case the functionality of the D-FF is given in a separate SystemC description included in this structural shift register through the #include "dff.h" header.

---

**Example:** Re-write the same 4-bit shift register in behavioral mode using the least number of lines of code.

**Solution:**

**Listing 8.12**: shiftreg_behav.h

```
// shiftreg_behav.h
#include <systemc.h>

SC_MODULE(shiftreg_behav) {
    sc_in<bool> clk;
    sc_in<bool> din;

   sc_out<sc_uint<4> > dout;
    // methods
    void count();

    //Constructor
    SC_CTOR( shiftreg_behav ) {
    SC_CTHREAD(count);
    sensitive << clk.pos();
    }
};
```

**Listing 8.13**: shiftreg_behav.cpp

```
// shiftreg_behav.cpp
#include "systemc.h"
#include "shiftreg_behav.h"

void shiftreg_behav::count()
```

```
{
    sc_uint<4> data;
    int x=0;
    dout.write(0);

    while (1) {
        wait(); // wait for any signal on sensitivity list
        for(x=3;x<0;x++)
            data[x] = data[x-1];
        data[0] = din.read();
        dout.write(data);
    }
}
```

It is very important for everyone writing SystemC programs to understand how to write a testbench and monitor certain signals by dumping them onto a VCD file.

**Example:** Write a testbench for the behavioral shift register just written creating a 20 clock pulses and random inputs to the input. Dump to a VCD file the clock, input of shift register and output.

**Solution:**

```
#include "systemc.h"
#include "shiftreg_behav.cpp"

int sc_main (int argc, char* argv[]) {
   sc_signal<bool>   clock;
   sc_signal<bool>   indata;
    sc_signal<sc_uint<4> > counter_out;
    int i = 0;
   // Connect the DUT
    shiftreg_behav counter("COUNTER");
   counter.clk(clock);
    counter.din(indata);
    counter.dout(counter_out);

  sc_start(1);
  // Open VCD file
  sc_trace_file *wf = sc_create_vcd_trace_file("counter");
  // Dump the desired signals
  sc_trace(wf, clock, "clock");
    sc_trace(wf, indata, "Data in");
  sc_trace(wf, counter_out, "Counter");

cout << "@" << sc_time_stamp() <<" Asserting Enable\n" << endl;
  enable = 1;  // Assert enable
  for (i=0;i<20;i++) {
    clock = 0;
    sc_start(1);
    clock = 1;
    din = rand \% 2;
    sc_start(1);
  }

  cout << "@" << sc_time_stamp() <<" De-Asserting Enable\n" << endl;
  enable = 0; // De-assert enable

  cout << "@" << sc_time_stamp() <<" Terminating simulation\n" << endl;
  sc_close_vcd_trace_file(wf);
  return 0;// End simulation
}
```

**Listing 8.14**: sc_main example

## 8.7 Synthesizable SystemC

SystemC allows to create complex system-level transactional models. These enable the creation of complete virtual platforms, even including the analog portions of the system through SystemC's Analog Mixed Signal class (AMS) .

In this book we are mainly interested in the synthesizable part of SystemC that allows HLS to generate a hardware module from the SystemC description. That good news is that this reduces the complexity of the SystemC making it very straight forward to write synthesizable SystemC. OSCI, now Accellera created a SystemC synthesizable subset draft [26] available at [2]. The latest draft has been under public review since 2015 (SystemC Synthesizable Subset Version 1.4).

2: https://www.accellera.org/downloads/drafts-review

Because I participated in drafting this latest draft, our lab created a benchmark suite of synthesizable SystemC benchmarks from different domains that comply with this draft. We call this benchmark suite S2CBench:Synthesizable SystemC Benchmarks. The next section describes these benchmarks in detail.

It should be noted that the draft does not explain how to synthesize the SystemC description. This is left to the individual HLS tool companies. Its main focus is on what should be synthesizable and what not.

The main restrictions that apply to synthesizable C/C++ also apply to SystemC. The code cannot have any dynamic data structures, and no recursion. Here are a few pitfalls that you need to avoid when writing synthesizable SystemC code.

**Pitfall 1:** The initialization part of the code in the main method (portion of code from the method start until the infinite while-loop in the sc_cthread wll be synthesized as the reset state. This implies that the code placed in this portion of the SystemC description needs to be synthesizable into hardware that is executed in one clock cycles. The main problem arises when the code has arrays synthesized as memories that want to be initialized.

Listing 8.15 shows a snippet of the reset portion of the of the ave8 benchmark. In this example the buffer[] array is initialized to 0. If buffer is synthesized as a memory, this would give you an error. The shown behavior could only be valid if the buffer[] array is expanded into individual FFs or synthesized as registers.

**Listing 8.15**: reset state pitfall

```
1  void ave8::ave8_main ( void ) {
2    //  Reset state: Needs to be executable in 1 clock cycle
3    sc_uint<11> sum=0;
4    sc_uint<8> buffer[SIZE];
5    int i;
6    // Initialize array
7   for(i=0;i<SIZE; i++)
       buffer[i] = 0;
8    wait();//  End Reset
9    while (1) {
10     :      :
11
12 }
```

**Pitfall 2:** Only sc_method and sc_cthreads are synthesizable. Sc_threads are more flexible that sc_cthreads, but it is not possible to generate a sequential circuit without a clock with commercial HLS tools. Thus, sc_threads are not synthesizable.

**Pitfall 3:** The synthesizable subset draft mentions that floating-point variables are non-synthesizable. The main reason for this is that synthesis of the floating-point data types used in C++ presents verification challenges. The simulation of the generated hardware has to be compared to the C++ behavior running in a processor. While most platforms conform to the IEEE 754 standard, this is not sufficient as the IEEE 754 standard allows different implementations to produce different results [26] . Having said this, most commercial HLS tools do synthesize floating-point variables. Please read their respective manuals.

## 8.8 Synthesizable SystemC Benchmarks (S2CBench)

S2CBench are available online at gitbhub.[3]. The initial release of S2CBench benchmark suite consisted in 2014 of 14 designs from different domains like DSP, image processing and cryptography [24]. The latest version (2.1) contains 19 designs including ANNs.

3: https://github.com/DARClab-UTD/S2CBench

The structure of all of the benchmarks is the same and similar to the two-module example shown in Figure 8.5, where the testbench is equivalent to module2 and the synthesizable module is module1. The testbench contains two threads. One thread sends data while the other thread receives the data from the synthesizable module. Once all of the test vectors read from a file are passed to the module, the simulation exists, but first checks if the outputs match the golden outputs stored in another external file.

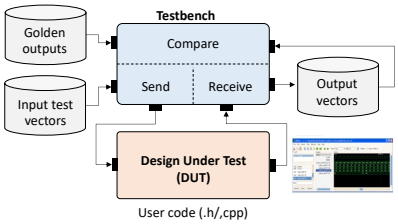



**Figure 8.7:** S2CBench benchmarks structure overview.

Figure 8.7 shows a block diagram of the benchmarks' structure. It should be noted that by default the benchmarks do not generate a VCD file. This can be enabled by recompiling the benchmark with the option -DWAVE in the compiler.

Listing 8.16 and listing 8.17 show an example of one of the benchmarks, in particular the one that computes the moving average of 8 numbers.

```
// ave8.h
#include <systemc.h>
#define SIZE 8

SC_MODULE (ave8) {
public:
   // Inputs
   sc_in_clk clk;
   sc_in<bool> rst;
   sc_in<sc_uint<8> > indata ;

   // Output
   sc_out< sc_uint<8> > ave8out;

  // Functions
  void ave8_main (void);

   // Constructor
 SC_CTOR (ave8) {
       SC_CTHREAD (ave8_main,
    clk.pos() );
       reset_signal_is(rst,
    false) ;
       sensitive << clk.pos();
  }
   // Destructor
   ~ave8() {}
};
```

**Listing 8.16**: ave8.h

**Listing 8.17**: ave8.cpp

```
// ave8.cpp
#include "ave8.h"   // include entity

//  Main thread
void ave8::ave8_main ( void ) {
  //  Reset state: Needs to be executable in 1 clock cycle
  sc_uint<11> sum=0;
  sc_uint<8> buffer[SIZE];
  int i;
  wait();//  End Reset

  while (1) {
    // Shift data
    for(i=SIZE-1; i>0 ;i--)
       buffer[i]=buffer[i-1];

    // Read new data
    buffer[0] = indata.read();

    // Add values together
    sum=buffer[0];
    for(i=1;i < SIZE ; i++)
         sum += buffer[i];

    // computer average
    ave8out.write(sum/SIZE) ;
      wait();
    }
}
```

## 8.9 Conclusions

This chapter has covered SystemC. Hopefully by the end of the chapter you will understand why SystemC is so powerful. The additional features that SystemC has compared to pure ANSI-C or C++ makes allows hardware designers to easier model and control the final generated hardware circuit. SystemC is nevertheless not popular in in the FPGA world. The main reason is that it is also more difficult to learn and it contains too many low-level details that many FPGA users to not care about.

FPGA vendors are trying to cater their FPGA to software engineers with limited hardware design skills. For them SystemC is still too low level as it has many similitudes to HDLs.

**Summary**

- SystemC is a C++ class for hardware design.
- SystemC is very popular in ASIC design with all ASIC HLS tools supporting its synthesizable subset.
- SystemC is basically a wrapper around C/C++ descriptions. Its main advantages are its custom data types (including fixed-point) and the ability to model concurrency.
- SystemC models concurrent behavior through multiple threads and a central scheduler. Wait() statements added by the user manually passes the control of the execution to the scheduler to switch to another thread.

# Commercial HLS tools 9

## 9.1 Introduction

There are a variety of different commercial HLS tools. These can mainly be classified into tools that mainly target ASICs or FPGAs. ASICs are Application Specific Integrated Circuits, basically, custom ICs build from scratch. FPGAs on the other hand are Field-Programmable Gate Arrays. Pre-manufactured ICs that only need to be configured. These FPGAs have different type on-chip resources like Look-up tables (LUTs) onto where any combinational logic function can be mapped to, embedded RAM (often called BlockRAM) and embedded DSP macros (basically embedded multipliers or multiply-accumulated units).

Table 9.1 highlights the main commercial HLS tools available at the writing of this book classified in either ASIC [20, 74, 75] or FPGA [16, 72, 76]. The table also shows the input languages supported by those tools. It can be observed that all of the ASIC tools support SystemC as input language while in the FPGA case it is mainly C and C++.

We should also note that there are multiple academic HLS tools available for anyone interested in *looking under the hood* of a HLS tool. Table 9.2 highlights some of them [77, 78].

It should be noted that two of the commercial HLS tools listed in table 9.1 first started as academic tools that were later span-off into companies later acquired. In particular, Xilinx Vivado HLx was a spin-off of Prof. Jason Cong's laboratory at UCLA, and Microchip SmartHLS Compiler a spin-off of Prof. Jason Anderson laboratory at Toronto University. The first was a tool called AutoPilot and the startup name AutoESL [79], while the latter's tool name was LegUP and the company name LegUP computing [17].



[20]: NEC (2022), *CyberWorkBench*
[74]: Cadence (2022), *Stratus*
[75]: Mentor (2022), *Catapult*
[16]: AMD/Xilinx (2022), *Vivado HLx*
[72]: Microchip (2022), *SmartHLS*
[76]: Intel (2022), *HLS Compiler*

| | HLS vendor tool name | Input Languages |
|---|---|---|
| ASIC | Cadence Stratus | C/C++/SystemC/MATLAB |
| | NEC CyberWorkBench | C/BDL/C++/SystemC |
| | Siemens EDA Catapult | C++/SystemC |
| FPGA | AMD/Xilinx Vitis HLS | C/C++ |
| | Intel HLS Compiler | C++ |
| | MATLAB HDL coder | M languages |
| | Microchip SmartHLS Compiler | C/C++ |

**Table 9.1:** Overview of Commercial HLS tools, their supported input languages.

**Table 9.2:** Overview of Academic HLS tools.

| Target HW Platform | Developer Tool name | Input a Languages |
|---|---|---|
| ASIC/FPGA | Politecnico de Milano, BAMBU | C/C++ |
| ASIC/FPGA | Université Bretagne Sud, GAUT | C/C++ |

## 9.2 AMD/Xilinx Vitis HLS

One of the most popular HLS tools are Vitis HLS from AMD/Xilinx. The precursor of Vitis HLS was developed by Prof. Jason Cong and his graduate students at UCLA and was called AutoPilot [79] that was then span off into AutoESL and later acquired by Xilinx and renamed into Vivado HLS, then Vivado HLx and recently renamed again to Vitis HLS.

Vitis HLS synthesizes a C/C++ into RTL code for implementation on a Xilinx FPGA or the reconfigurable fabric of a configurable SoC FPGA like Zynq FPGA. Vitis HLS is tightly integrated Xilinx's Vivado Design Suite that is responsible for further processing the design through synthesis, place, and route, and bitstream generation.

Vitis HLS takes as input ANSI-C or C++ and generates RTL code that can then be packaged into an IP and then passed on to different Xilinx tools to be integrated either in the larger hardware system (Vivado Design Suite) or with the embedded processor (Xilinx Platform studio).

Vitis HLS does traditional HLS and also allows to fully very through behavioral simulations or RTL simulations. It also makes extensive use of synthesis directives in the form of pragmas.

## 9.3 Intel HLS Compiler

Intel has made several efforts to make it easier to target their FPGAs, starting with an SDK to synthesize OpenCL. They later released a traditional HLS synthesizer that can synthesize ANSI-C. The next subsections describe these efforts.

### 9.3.1 FPGA SDK for OpenCL

Altera first entered the HLS world with their FPGA SDK for OpenCL. The main goal of this tool was to synthesize OpenCL into their FPGAs.

This synthesizer does not perform a traditional HLS as described in this book, but maps the OpenCL code to an architectural template that can be tailored through the OpenCL code and synthesis options.

This flow is more tailored to designers that have very little hardware design knowledge and want to exploit the FPGAs' ability to accelerate computationally intensive applications through a CPU+FPGA computer.

### 9.3.2 HLS Compiler

HLS compiler is a more recent effort developed in-house by Intel to synthesize ANSI-C/C++ descriptions into RTL. This is a traditional HLS tool as described during this text book. It makes use of extensive synthesis directives like most commercial HLS tools and is tightly integrated with Intel's Quartus Prime design flow to generate the bitstream for their FPGAs.

The HLS compiler flow resembles a traditional HLS flow that allows for architectural exploration by modifying the different HLS options. Users can first iterate through the pure SW description of the application until the functionality is correct and then re-iterate again the architectural to obtain the desired HW architecture.

Once you have generated the desired architecture which meets your constraints, you can pass it to Intel's main FPGA programming flow (Quartus Prime) through another tool called Qsys which integrates different components in the system together.

## 9.4 MATLAB HDL Coder

Although MATLAB is not known for being an EDA vendor, but they do provide a synthesis tool that can map MATLAB code directly into an FPGA called HDL coder.

HDL coder converts either MATLAB code or Simulink model into RTL. Similarly to Intel's FPGA SDK for OpenCL, this tool does not perform traditional HLS, but basically translates the Matlab code into a parametrizable template-based architecture.

## 9.5 MicroChip SmartHLS Compiler

Microchip recently acquired Legup computing, a spinoff Toronto University that created the an open-source HLS tool called LegUp [17]. and rebranded the HLS tool as SmartHLS compiler. the flow is very similar to the Intel and AMD/Xilinx flow.

SmartHLS is an eclipse-based integrated development environment that takes C++ descriptions as input and generates an IP component (Verilog HDL) as output that can then be passed on to the FPGA design flow.

## 9.6 Cadence Stratus

Cadence Stratus is one of the last ASIC HLS tool to enter the market. In 2008 Cadence released a tool called C-to-Silicon Compiler. With the acquisition of Forte Design System and its Cynthesizer HLS tool in 2014 the company rebranded the tool to Stratus.

Stratus supports SystemC and recently also started supporting MATLAB as input language.

## 9.7 NEC CyberWorkBench

NEC developed since the 90's a HLS tools called CyberWorkBench (CWB). Initially it was it was developed as an in-house tool for the semiconductor division and since early 2000's it was made commercially available to outside customers.

The core of the tools is the traditional HLS engine, but it is basically a design environment to design complete ICs at the behavioral level. CWB can target ASICs and FPGAs and supports ANSI C, C++, SystemC and BDL (NEC's C language with HW extensions).

## 9.8 Siemens Catapult

Catapult is one of the oldest HLS tools. Developed in-house at Mentor and released in 2004 as Catapult C. Initially it only supported C++ and later on added SystemC.

The overall flow, which is similar to the other HLS tools providing a main HLS engine and different optimization and verification alternatives.

## 9.9 Summary

This chapter tries to help you navigate around all the HLS options available to you at the writing of this book. There are pros and cons of every tool and it is important that you fully understand them. ASIC tools are very powerful and allow many customizations, but this also implies steeper learning curves. FPGA tools are simple to use and more importantly basically free. One of the problem with FPGA tools is that you can mostly only use them for the FPGA vendor's own FPGA and hence they lack portability.

**Summary**

- There are many commercial HLS tools available.
- FPGA tools are free, but limited to the FPGAs of that particular vendor. ASIC tools are expensive but more flexible as they can target any ASIC technology and FPGA.
- ASIC tools give you very low-level controllability allowing you to generate virtually any circuit. The drawback is the steeper learning curve.

# Appendix

# Logic Synthesis Scripts 10

**Synopsys Design Compiler**

```
#  Reading in RTL file}
analyze -f verilog hls_rtl.v
elaborate hls_rtl

#  Setting up constraints
current_design hls_rtl
create_clock CLOCK -name CLOCK -period 10
set_input_delay -clock CLOCK -rise 0.0 [all_inputs]
set_output_delay -clock CLOCK -rise 0.0 [all_outputs]
set_max_fanout 10 hls_rtl
set_max_transition 2.5 hls_rtl
set_ideal_network RESET
set_ideal_network CLOCK
set_auto_disable_drc_nets -constant false -clock true

#  logic optimization
check_design
link
uniquify
compile -boundary_optimization -map_effort medium
compile -incremental_mapping -map_effort medium

#  writing out files
write -f ddc -hier -o hls_rtl_gate.ddc
write -f verilog -hier -o hls_rtl_gate.v
write_sdc hls_rtl_gate.sdc

#  generating report files
redirect hls_rtl_gate_net.rpt { report_net }
redirect hls_rtl_gate_timing.rpt { report_timing }
redirect hls_rtl_gate_area.rpt { report_area }
redirect hls_rtl_gate_qor.rpt { report_qor }
redirect hls_rtl_gate_vol.rpt { report_constraint -all_violators }

quit
```

**AMD/Xilinx Vivado**

```
create_project -part XC7V2000T-FHG1761-1 -force hls_rtl
set_property target_language Verilog [current_project]
add_files -norecurse hls_rtl.v
add_files -norecurse hls_rtl.tcl.xdc

# IP_Catalog
launch_runs synth_1
wait_on_run synth_1
launch_runs impl_1
wait_on_run impl_1
```

**hls_rtl.tcl.xdc**

```
create_clock CLOCK -name CLOCK -period 10.000000
```

**Intel Quartus II**

```
load_package flow
project_new hls_rtl -overwrite
set_global_assignment -name VERILOG_FILE hls_rtl.v
set_global_assignment -name DEVICE 5SEE9F45C2
set_global_assignment -name FAMILY stratixV
set_global_assignment -name TOP_LEVEL_ENTITY ave16
set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
set_global_assignment -name SDC_FILE hls_rtl.sdc
execute_module -tool map
execute_module -tool fit
execute_module -tool sta -args "--multicorner=on --report_script=hls_rtl.sta.tcl"
project_close
```

**hls_rtl.sta.tcl**

```
create_clock CLOCK -name CLOCK -period 10.000000
```

# RTL Simulation Scripts 11

**Siemens Modelsim**

```
vsim -c -do vr_make.do
```

**vr_make.do**

```
quit -sim
vlib work

######################
# Compile
######################
vcom -explicit -93 +acc hls_rtl.vhd

######################
# Test Bench library
######################
vcom -explicit -93 +acc tb1.vhd
vcom -explicit -93 +acc tb2.vhd


######################
# Test Bench
######################
vcom -explicit -93 +acc hls_tb.vhd

######################
# Simulation
######################
vsim work.T_hls_rtl_00 -t ps -wlf waveform_out.wlf

view structure
view signals
view wave

do ./vh_signal.do

set StdArithNoWarnings 1
set WaveSignalNameWidth 2
set DefaultRadix hex

run -all

quit
```

**vh_signals.do**

```
add wave -noupdate -divider {test signal}
add wave -noupdate -format Logic /TB_HLS/sig1
```

```
add wave -noupdate -format Literal /TB_HLS/sig2
add wave -noupdate -divider {system clock , system reset}
add wave -noupdate -format Logic /T_hls_RTL/U_UUT/CLOCK
add wave -noupdate -format Logic /T_hls_RTL/U_UUT/RESET
```

**Synopsys VCS**

```
vcs +v2k hls_rtl_tb.v hls_rtl.v
./simv | tee simv.log
```

**Icarus Verilog**

```
iverilog hls_rtl_tb.v hls_rtl.v
vvp a.out
```

**AMD/Xilinx Vivado**

```
xvlog hls_rtl_tb.v hls_rtl.v
xelab T_hls_top_mod -s run_sim --timescale 1ps/1ps
xsim -R run_sim
```

**Aldec Riviera**

```
vsimsa -do vr_run.tcl
```

**vr_run.tcl**

```
transcript file transcript
quit -sim
alib work

######################
# Compile
######################
alog +acc -incr hls_rtl_tb.v hls_rtl.v

######################
# Simulation
######################
asim +access +r work.T_hls_top_mod
log -rec /*
run -all
quit
```

# Bibliography

References in citation order.


[1] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. 1st. Springer Publishing Company, Incorporated, 2008 (cited on pages 5, 39).

[2] Renesas Electronics. *Stream Transpose Processor*. July 2020. URL: https://www.renesas.com/us/en/products/power-management/pmic/stp-engine.html (cited on page 20).

[3] Changmoo Kim et al. 'ULP-SRP: Ultra low power Samsung Reconfigurable Processor for biomedical applications'. In: *2012 International Conference on Field-Programmable Technology* (2012), pp. 329–334 (cited on page 20).

[4] P.G. Paulin and J.P. Knight. 'Force-directed scheduling for the behavioral synthesis of ASICs'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.6 (1989), pp. 661–679. DOI: 10.1109/43.31522 (cited on page 25).

[5] Rajiv Jain et al. 'Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics'. In: *Proceedings of the 28th ACM/IEEE Design Automation Conference*. DAC '91. San Francisco, California, USA: Association for Computing Machinery, 1991, pp. 686–689. DOI: 10.1145/127601.127751 (cited on page 25).

[6] J. Cong and Zhiru Zhang. 'An efficient and versatile scheduling algorithm based on SDC formulation'. In: *Design Automation Conference*. 2006, pp. 433–438 (cited on pages 25, 27).

[7] S. Raje and R. A. Bergamaschi. 'Generalized Resource Sharing'. In: *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '97. San Jose, California, USA: IEEE Computer Society, 1997, pp. 326–332 (cited on page 45).

[8] Stefan Hadjis et al. 'Impact of FPGA Architecture on Resource Sharing in High-level Synthesis'. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '12. Monterey, California, USA: ACM, 2012, pp. 111–114. DOI: 10.1145/2145694.2145712 (cited on pages 45, 55).

[9] B. Carrion Schafer and Z. Wang. 'High-Level Synthesis Design Space Exploration: Past, Present, and Future'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2628–2639 (cited on pages 53, 58).

[10] F. Ferrandi et al. 'A Multi-objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis'. In: *2008 IEEE Computer Society Annual Symposium on VLSI*. 2008, pp. 417–422. DOI: 10.1109/ISVLSI.2008.73 (cited on page 54).

[11] Gang Wang et al. 'Design space exploration using time and resource duality with the ant colony optimization'. In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006, pp. 451–454. DOI: 10.1145/1146909.1147028 (cited on page 54).

[12] Byoungro So, Mary W. Hall, and Pedro C. Diniz. 'A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems'. In: *SIGPLAN Not.* 37.5 (May 2002), pp. 165–176. DOI: 10.1145/543552.512550 (cited on page 54).

[13] Robert Wilson et al. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*. Tech. rep. Stanford, CA, USA, 1994 (cited on page 54).

[14] S. Bakshi and D. D. Gajski. 'Component selection for high-performance pipelines'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.2 (1996), pp. 181–194. DOI: 10.1109/92.502191 (cited on page 54).

[15] W. Sun, M. J. Wirthlin, and S. Neuendorffer. 'FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 254–265. DOI: 10.1109/TCAD.2006.887923 (cited on page 54).

[16] AMD/Xilinx. *Vivado HLx*. 2022. URL: www.xilinx.com (cited on pages 54, 72, 99).

[17] Andrew Canis et al. 'LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems'. In: *ACM Trans. Embed. Comput. Syst.* 13.2 (2013). DOI: 10.1145/2514740 (cited on pages 54, 99, 101).

[18] Jason Cong et al. 'A Study on the Impact of Compiler Optimizations on High-Level Synthesis'. In: *Languages and Compilers for Parallel Computing*. Ed. by Hironori Kasahara and Keiji Kimura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 143–157 (cited on page 54).

[19] Q. Huang et al. 'The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs'. In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 2013, pp. 89–96. DOI: 10.1109/FCCM.2013.50 (cited on page 54).

[20] NEC. *CyberWorkBench*. 2022. URL: www.cyberworkbench.com (cited on pages 55, 99).

[21] Jason Cong and Wei Jiang. 'Pattern-based Behavior Synthesis for FPGA Resource Reduction'. In: *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. FPGA '08. Monterey, California, USA: ACM, 2008, pp. 107–116. DOI: 10.1145/1344671.1344688 (cited on page 55).

[22] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. 'Accelerating FPGA Prototyping Through Predictive Model-Based HLS Design Space Exploration'. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: ACM, 2019, 97:1–97:6. DOI: 10.1145/3316781.3317754 (cited on page 55).

[23] Yuko Hara et al. 'CHStone: A benchmark program suite for practical C-based high-level synthesis'. In: *2008 IEEE International Symposium on Circuits and Systems*. May 2008, pp. 1192–1195. DOI: 10.1109/ISCAS.2008.4541637 (cited on page 55).

[24] B. Carrion Schafer and A. Mahapatra. 'S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis'. In: *IEEE Embedded Systems Letters* 6.3 (Sept. 2014), pp. 53–56 (cited on pages 55, 97).

[25] IEEE Computer society. *IEEE Standard for Standard SystemC Language Reference Manual*. Tech. rep. 2012 (cited on pages 55, 79, 83).

[26] Accelera. *SystemC Synthesizable Subset Version 1.4.7*. Tech. rep. 2016 (cited on pages 55, 79, 83, 96, 97).

[27] B. Khailany et al. 'INVITED: A Modular Digital VLSI Flow for High-Productivity SoC Design'. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. June 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465897 (cited on page 55).

[28] Q. Gautier et al. 'Spector: An OpenCL FPGA benchmark suite'. In: *2016 International Conference on Field-Programmable Technology (FPT)*. Dec. 2016, pp. 141–148 (cited on page 55).

[29] Dong Liu and B. Carrion Schafer. 'Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs'. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–8 (cited on page 56).

[30] Steve Dai et al. 'Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning'. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2018), pp. 129–132 (cited on page 56).

[31] Kalyanmoy Deb et al. 'A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II'. In: *Parallel Problem Solving from Nature PPSN VI*. Ed. by Marc Schoenauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 849–858 (cited on page 56).

[32] D. A. Van Veldhuizen and G. B. Lamont. 'On measuring multiobjective evolutionary algorithm performance'. In: *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*. Vol. 1. July 2000, 204–211 vol.1. DOI: 10.1109/CEC.2000.870296 (cited on page 56).

[33] E. Zitzler et al. 'Performance assessment of multiobjective optimizers: an analysis and review'. In: *IEEE Transactions on Evolutionary Computation* 7.2 (Apr. 2003), pp. 117–132. DOI: 10.1109/TEVC.2003.810758 (cited on page 56).

[34] E. Zitzler et al. 'Performance assessment of multiobjective optimizers: an analysis and review'. In: *IEEE Transactions on Evolutionary Computation* 7.2 (Apr. 2003), pp. 117–132. DOI: 10.1109/TEVC.2003.810758 (cited on page 56).

[35] B. Carrion Schafer. 'Probabilistic Multi-knob High-Level Synthesis Design Space Exploration Acceleration'. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* PP.99 (2015), p. 1 (cited on pages 58, 59).

[36] B. Carrion Schafer and Kazutoshi Wakabayashi. 'Design Space Exploration Acceleration Through Operation Clustering'. In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 29.1 (Jan. 2010), pp. 153–157 (cited on pages 58, 60).

[37] B. Carrion Schafer and K. Wakabayashi. 'Machine learning predictive modelling high-level synthesis design space exploration'. In: *IET Computers Digital Techniques* 6.3 (May 2012), pp. 153–159 (cited on pages 58, 60, 61).

[38] A. Mahapatra and B. C. Schafer. 'Machine-learning based simulated annealer method for high level synthesis design space exploration'. In: *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*. 2014, pp. 1–6 (cited on page 58).

[39] B. Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 'Adaptive Simulated Annealer for high level synthesis design space exploration'. In: *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*. Apr. 2009, pp. 106–109 (cited on pages 58, 59).

[40] Benjamin Carrion Schafer. 'Parallel High-Level Synthesis Design Space Exploration for Behavioral IPs of Exact Latencies'. In: *ACM Trans. Des. Autom. Electron. Syst.* 22.4 (May 2017), 65:1–65:20 (cited on page 59).

[41] Scott Kirkpatrick. 'Optimization by simulated annealing: Quantitative studies'. In: *Journal of Statistical Physics* 34.5 (1984), pp. 975–986 (cited on page 59).

[42] Vipul Kumar Mishra and Anirban Sengupta. 'PSDSE: Particle Swarm Driven Design Space Exploration of Architecture and Unrolling Factors for Nested Loops in High Level Synthesis'. In: *2014 Fifth International Symposium on Electronic System Design* (2014), pp. 10–14 (cited on page 59).

[43] Anirban Sengupta and Saumya Bhadauria. 'Exploration of Multi-objective Tradeoff during High Level Synthesis Using Bacterial Chemotaxis and Dispersal'. In: *18th International Conference in Knowledge Based and Intelligent Information and Engineering Systems, KES 2014, Gdynia, Poland, 15-17 September 2014*. 2014, pp. 63–72 (cited on page 59).

[44] B. Carrion Schafer and Kazutoshi Wakabayashi. 'Divide and Conquer High-level Synthesis Design Space Exploration'. In: *ACM Trans. Des. Autom. Electron. Syst.* 17.3 (July 2012), 29:1–29:19 (cited on page 60).

[45] S. Xydis et al. 'Efficient High Level Synthesis Exploration Methodology Combining Exhaustive and Gradient-Based Pruned Searching'. In: *ISVLSI*. July 2010, pp. 104–109 (cited on page 60).

[46] N. K. Pham et al. 'Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis'. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. New York, NY, USA: ACM, 2014 (cited on page 60).

[47] Alessandro Cilardo and Luca Gallo. 'Interplay of Loop Unrolling and Multidimensional Memory Partitioning in HLS'. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE '15. Grenoble, France: EDA Consortium, 2015, pp. 163–168 (cited on page 60).

[48] Luca Piccolboni et al. 'COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators'. In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017) (cited on page 60).

[49] Cody Hao Yu et al. 'S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters'. In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California: ACM, 2018, 153:1–153:6. DOI: 10.1145/3195970.3196109 (cited on page 60).

[50] L. Ferretti, G. Ansaloni, and L. Pozzi. 'Cluster-Based Heuristic for High Level Synthesis Design Space Exploration'. In: *IEEE Transactions on Emerging Topics in Computing* (2018), pp. 1–9 (cited on page 60).

[51] Giovanni Ansaloni Lorenzo Ferretti and Laura Pozzi. 'Lattice-Traversing Design Space Exploration for High Level Synthesis'. In: *2018 IEEE International Conference on Computer Design (ICCD)*. Nov. 2018, pp. 509–512 (cited on page 60).

[52] A. Prost-Boucle, O. Muller, and F. Rousseau. 'A Fast and Autonomous HLS Methodology for Hardware Accelerator Generation under Resource Constraints'. In: *2013 Euromicro Conference on Digital System Design*. 2013, pp. 201–208. DOI: 10.1109/DSD.2013.30 (cited on page 60).

[53] G. Palermo, C. Silvano, and V. Zaccaria. 'ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.12 (Dec. 2009), pp. 1816–1829. DOI: 10.1109/TCAD.2009.2028681 (cited on page 60).

[54] Marcela Zuluaga et al. '"Smart" Design Space Sampling to Predict Pareto-optimal Solutions'. In: *SIGPLAN Not.* 47.5 (June 2012), pp. 119–128 (cited on page 60).

[55] Hung-Yi Liu and L.P. Carloni. 'On learning-based methods for design-space exploration with High-Level Synthesis'. In: *DAC*. May 2013, pp. 1–7 (cited on page 60).

[56] S. Xydis et al. 'SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.1 (Jan. 2015), pp. 155–159. DOI: 10.1109/TCAD.2014.2363392 (cited on page 60).

[57] Guanwen Zhong et al. 'Design space exploration of multiple loops on FPGAs using high level synthesis'. In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)* (2014), pp. 456–463 (cited on page 61).

[58] G. Zacharopoulos et al. 'Machine Learning Approach for Loop Unrolling Factor Prediction in High Level Synthesis'. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. July 2018, pp. 91–97. DOI: 10.1109/HPCS.2018.00030 (cited on page 61).

[59] G. Zhong et al. 'Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators'. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2016, pp. 1–6 (cited on page 61).

[60] J. Zhao et al. 'COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications'. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 430–437 (cited on page 61).

[61] Shuo Wang, Yun Liang, and Wei Zhang. 'FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs'. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: ACM, 2017, 27:1–27:6. DOI: 10.1145/3061639.3062251 (cited on page 61).

[62] S. Kurra, N. K. Singh, and P. R. Panda. 'The Impact of Loop Unrolling on Controller Delay in High Level Synthesis'. In: *2007 Design, Automation Test in Europe Conference Exhibition*. Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364623 (cited on page 62).

[63] Z. Wang and J. Chen and B. Carrion Schafer. 'Efficient and Robust High-Level Synthesis Design Space Exploration through offline Micro-kernels Pre-characterization'. In: *DATE*. 2020, pp. 145–150 (cited on page 62).

[64] L. Ferretti, J. Kwon, G. Ansaloni, G. Di Guglielmo, L. P. Carloni and L. Pozzi. 'Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis'. In: *IEEE TCAD* (Nov. 2020), pp. 1–12 (cited on page 62).

[65] Zi Wang and , Benjamin Carrion Schafer. 'Learning from the Past: Efficient High-Level Synthesis Design Space Exploration for FPGAs'. In: *ACM Trans. Des. Autom. Electron. Syst.* 27.4 (2022) (cited on page 62).

[66] Md. Imtiaz Rashid and B. Carrion Schafer. 'Fast and Inexpensive High-Level Synthesis Design Space Exploration : Machine Learning to the Rescue'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–13 (cited on page 62).

[67] Scott Kirkpatrick. 'Optimization by simulated annealing: Quantitative studies'. In: *Journal of Statistical Physics* 34.5 (Mar. 1984), pp. 975–986 (cited on page 63).

[68] Z. Wang and B. Carrion Schafer. In: (cited on page 66).

[69] Md Imtiaz Rashid and Benjamin Carrion Schafer. 'Fast Parallel High-Level Synthesis Design Space Explorer: Targeting FPGAs to Accelerate ASIC Exploration'. In: *Proceedings of the Great Lakes Symposium on VLSI 2022*. GLSVLSI '22. Irvine, CA, USA: Association for Computing Machinery, 2022, pp. 85–90. DOI: 10.1145/3526241.3530339 (cited on page 67).

[70] DARClab. *High-Level Synthesis Design Space Exploration Framework*. 2019 (cited on pages 67, 70).

[71] Md Imtiaz Rashid and Benjamin Carrion Schaefer. 'Improving the Quality of Hardware Accelerators through automatic Behavioral Input Language Conversion in HLS'. In: *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2022, pp. 623–628. DOI: 10.1109/ASP-DAC52403.2022.9712582 (cited on page 72).

[72] Microchip. *SmartHLS*. 2022. URL: https://microchiptech.github.io/fpga-hls-docs/ (cited on pages 72, 99).

[73] Mentor. *Algorithmic C Data Types*. URL: https://github.com/hlslibs/ac_types (cited on page 73).

[74] Cadence. *Stratus*. 2022. URL: www.cadence.com (cited on pages 80, 99).

[75] Mentor. *Catapult*. 2022. URL: www.mentor.com (cited on page 99).

[76] Intel. *HLS Compiler*. 2022. URL: www.intel.com (cited on page 99).

[77] C. Michael Pilato and Fabrizio Ferrandi. 'Bambu : A Free Framework for the High Level Synthesis of Complex Applications'. In: 2012 (cited on page 99).

[78] Philippe Coussy et al. 'GAUT: A High-Level Synthesis Tool for DSP Applications'. In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 147–169 (cited on page 99).

[79] Zhiru Zhang et al. 'AutoPilot: A Platform-Based ESL Synthesis System'. In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 99–112 (cited on pages 99, 100).

# Alphabetical Index

**High-Level Synthesis Made Easy** **First Edition**

Synthesizing Behavioral Descriptions directly into Hardware Circuits

This book is intended to teach anyone, either someone with deep hardware development skills, or someone with little prior hardware design knowledge alike to learn how to design and verify dedicated hardware modules using HLS.

It covers the basic HLS steps, and also practical problems such as how to set the different HLS synthesis options to obtain the desired hardware or a dedicated.

## Features

- Includes an overview of different hardware platforms that HLS targets including different types of FPGAs
- Features a chapter dedicated to automatic HLS design space exploration to find the most optimal hardware implementations
- Offers code examples of how to write efficient SystemC code for HLS
- Review of modern HLS tools (ASIC and FPGA)

## About the Author

**Benjamin Carrion Schaefer**
Associate Professor of Electrical and Computer Engineering
The University of Texas at Dallas

The author has over 20 years of experience in developing academic and industrial HLS tools. He has published over 100 academic papers in this area, and has participated in writing the latest SystemC synthesizable subset draft.

Published by highX Technologies, 2023