Overview of high level synthesis tools

Spyridon Georgakakis\textsuperscript{a} and John Evans\textsuperscript{b,1}

\textsuperscript{a}Technological Educational Institute of Piraeus, School of technological applications, Department of Electronics Athens, Greece

\textsuperscript{b}CERN, Geneva, Switzerland

E-mail: john.evans@cern.ch

Abstract: High Level Synthesis takes an abstract behavioural or algorithmic description of a digital system and creates a register transfer level structure that realises the described behaviour. Various methodologies have been developed to perform such synthesis tasks. This paper presents the different HLS concepts used in the current leading tools. It makes a comparison between the different approaches and highlights their advantages and limitations. We also present a high level synthesis example.

Keywords: Digital electronic circuits; Data acquisition circuits; Front-end electronics for detector readout; Data acquisition concepts

\textsuperscript{1}Corresponding author.
1 Introduction

High level synthesis as an idea and its implementation has changed through the years. Initially an effort to automate the process of generating hardware from “behavioural” instead of “structural” descriptions, it has evolved as a technique to enhance the production of large integrated circuits and more complicated digital systems. The realization of this technical challenge continues to evolve as it tries to reach standardization and industry adoption.

HLS can presently be considered as consisting of 3 main tasks: system description, scheduling, resource allocation and binding. System description includes specifying the functionality of the eventual hardware, scheduling assigns the operations and their execution in time, and resource allocation and binding is used to define the implemented hardware. Due to the complexity of the tasks and their interdependence, HLS tool evolution has been rather difficult. For example, the structure of a behavioural description model can present significant problems of concurrency and hierarchy structure that ultimately needs to be managed by an efficient scheduling method.
These complexities gave rise to a set of problems that led to failure of some early HLS tools. These often resulted in poor quality of results and difficult validation of the implemented hardware. However, today’s commercial high level synthesis products claim to generate correct and high quality RTL faster than manual design methods in specific application domains. There has also been a change in input methods used for HLS with earlier behavioural HDLs being replaced in favour of other high-level languages more familiar to DSP and algorithm designers. However, it has also raised the issue of the expected target user. It has not been clear if these tools have been made for software engineers to produce hardware or for hardware engineers to be more involved in system design.

2 HLS input methods

2.1 High-level Languages

There are different methods of entering information using a mixture of text and graphical user interfaces. From the text-input side, methods are based mostly on high-level description languages where again there can be several variants. These can be split into two main divisions. One is based on essentially mainline high level languages while another approach is to use languages specifically developed with HLS in mind.

2.1.1 C variations

Applicable high-level languages consist mainly of C/C++ variations and supersets. It is easy to understand why these would be considered as a starting-point for HLS. The languages are mature, have been used for many years and have a range of associated tools such as debuggers and compilers available. Almost by definition, it is relatively easy to express algorithms in these languages and there are standard texts available that list many already previously written for specific fields e.g. digital signal processing. It is easy to incorporate different levels of abstraction into the modelling. Architecture design is traditionally done using these languages as the relatively fast simulation time allows quick and easy exploration of the design space.

C and each variation are sequential languages with no immediate construct available to describe concurrency: an idea that is inherent in hardware design. Language attributes such as weak typing and lack of garbage collection also suggest that they might not be suitable for HLS. These traits could make hardware design and HLS debugging difficult. In addition, the use of non-standard proprietary language extensions is needed to describe some hardware aspects like fixed-point arithmetic.

SystemC was conceived to try and overcome some of these limitations. Although it is a C++ special class and unavoidably inherits a lot C/C++ features, it allows hardware-based notions like concurrency and fixed point arithmetic to be included in a design. The perceived advantage of the solution is that any new or existing code written by a software developer could be modified to become possible to realize in hardware. There is an Open SystemC initiative where the latest LRM and OSCI SystemC kernel can be downloaded. There is also support available from an extensive online user community.
2.1.2 HLS specific languages

Some languages have been designed specifically for use in HLS. Bluespec is such an example. Through the adaptation of various features of high level languages (mainly Haskell) and HDL language characteristics (SystemVerilog), it aims to reduce the distance from behavioural model to RTL design. The language adopts the aspect of atomic transactions to express and realize complex concurrent behaviours. It offers the advantage of proposing a common unified language for architects, implementers and verification engineers.

However, the adoption of a new language and technology implies a steep learning curve and existing C-based algorithms need re-working before they can be used in these new flows.

2.2 MATLAB language and Simulink

MATLAB and especially Simulink have traditionally been extensively used for algorithm design. The availability of a mature, supported tool with specialized modules (toolboxes, blocksets) along with the possibility of integrating C-code makes the tool a very attractive development platform. However, while convenient for development, there still remained the problem of implementing the algorithms in hardware.

FPGA vendors have developed Simulink blocksets that allow pre-defined functions to be simply and efficiently synthesized into their devices. Examples include Altera DSP Builder and Xilinx SysGen. While easy to use, the tools are limited to the vendor-supplied blocks and are only available for proprietary devices. Some attempts were made to produce a more general tool that would allow MATLAB M-code to be directly synthesized but this more general approach has led to some failures. This approach runs into the same difficulties as using C or one of its variants as an input language, not least the one that means it is easy to produce code that cannot be synthesized.

However, at least one commercial product is currently available that attempts to overcome these limitations.

3 Main concepts of synthesis proposed by the tools

3.1 Untimed C/C++ (algorithmic approach)

High level of abstraction of the input source separates the untimed core computation from the hardware. Untimed models are easier to debug, verify and simulate. The generation of the hardware interface is left to the HLS tool which can be used to generate different architectures from the same input source.

Transactional Level Modelling, typically but not exclusively used with SystemC, separates the description of functional modules from the means and methods used to communicate between them. This allows a wide range of abstraction that can be used to maximize simulation speeds and to minimize architecture exploration times. Functionality and communication can gradually be refined to the eventual level of pin-cycle accuracy.

3.2 Timed C

Computation core (untimed C) can be used with a hardware level of abstraction (timed C, TLM). Algorithms are written at the transaction level and SystemC provides a link between
high level source code and low level implementation ensuring a representation that directly maps to RTL. The tool can be used for verification through a TLM block based manner and for optimizing RTL generation.

3.3 Unified synthesis

Algorithms are developed taking into consideration architecture and hardware possibilities from the outset. This is a different approach that needs a specialised language (with Bluespec being an example) able to describe both the algorithm and hardware constructs. It generally follows the idea of the timed approach.

4 Tool procedures

Although the approaches use different techniques, they tend to converge to basic ideas of high level synthesis. These include scheduling problems, allocation of resources and the binding of design with the hardware modules. The procedure is automated but the user can guide the tool with the input of constraints regarding the design.

4.1 Loop control or iteration scheduling

Loops and tasks can be handled in different ways providing the user with the ability to expose and manipulate parallelism appropriate for that design. Unrolling can be used to create additional copies of the hardware to implement the loop or task. Pipelining can be used to increase the throughput by initiating the next loop iteration or task before the current iteration is completed.

4.2 Schedule directives

Almost all tools allow the possibility to interact with scheduling, mainly through directives that control latency, cycle time, throughput and timing of whole or part of the design.

4.3 Resource allocation and binding

The tools provide means to allow the user to interact with the resource binding. Resource manipulation can include specific allocation of design input-output, specific memory architectures and use of IP accelerating libraries.

5 Verification and quality of results

Earlier tools suffered from variable quality of results while producing difficult to validate hardware. Verification has become an essential part of modern HLS tools and it might even be their most compelling advantage over more traditional design flows.

Verification comprises two stages. The first verifies the algorithmic correctness of the input code and the second validates the correct transformation of the design description to RTL. The first step is usually done through simulation while the second is accomplished either by simulation-based methods or by equivalence checking. The abstraction level of the input code can offer a very high execution speed of simulation compared to traditional hardware behavioral languages.
descriptions. For simulating and/or verifying the output, most of the tools provide an automated process to allow the test benches used for the input code simulation of the input code to be re-used to verify the output RTL. The importance of being able to verify the RTL code against the original reference design cannot be understated.

6 Example with Mentor Graphic’s CatapultC

The goal was to produce an equivalent for a sorting algorithm design originally described using VHDL. The design takes in 20 words of 16-bit width, sorts them comparing the 6 less significant bits and then outputs the 4 bigger words.

The C algorithm was constructed in such a way to enable the program to expose parallelism using a ranking algorithm with constant iteration boundaries [1].

```c
void sort(int in[20], int out[4]){
    for (int i=0; i<20; i++){
        int x = 0;
        for (int j=0; j<20; j++){
            if ((in[i] > in[j]) || (in[i] == in[j] && (i<j)))
                x ++;
        }
        if (x >= 16)
            out[(19-x)] = in[i];
    }
}
```

The code was then modified to meet hardware requirements and constraints (ports width, unused register elimination etc) before being treated with Mentor Graphic’s CatapultC [2].

```c
void sort(ac_int<16,false> in[20], ac_int<16,false> out[4]){
    for (ac_int<5,false> i=0; i<20; i++){
        ac_int<5,false> x = 0;
        for (ac_int<5,false> j=0; j<20; j++){
            if ((in[i].sle<6>(0) > in[j].sle<6>(0)) ||
                (in[i].sle<6>(0) == in[j].sle<6>(0) && (i<j)))
                x ++;
        }
        if (x >= 16)
            out[(19-x)] = in[i];
    }
}
```

The device used was an Altera Stratix II (EP2S90F1020I) device with a 40MHz target clock frequency. Resource types were specified for input and output ports and word width constrained to 16 bits. In order to accomplish parallelization of the design, the two inner loops were fully unrolled and a combinational logic directive used to match the output with the VDHL implementation. The design was scheduled with latency in favour of area. The produced RTL output (10 lines of C code produced 2300 lines of VHDL code) was then synthesized using Mentor Graphic’s Precision synthesizer and placed with Altera’s Quartus II.

A timing analysis between the original VHDL implementation and the CatapultC derived design produced similar results. The CatapultC design resulted in lower (better) worst-case point to point delays difference but at at the expense of larger resource usage.

The RTL output was verified against the C-code input using CatapultC. For completeness, the VHDL and CatapultC results were verified as being equivalent using formal verification analysis with Mentor Graphic’s FormalPro.
7 Conclusions

We have provided an overview of the current state of HLS tools. We have also shown how a commercially available tool has been able to provide a functionally equivalent output to a traditional VHDL-based design flow while producing comparable results as regards speed and FPGA resource usage.

The RTL produced in CatapultC was verified automatically against the input C-code with no manual intervention needed.

While still not able to provide a full turnkey solution, currently available HLS tools now seem sufficiently developed to be practical to use and could help reduce design and especially verification efforts.

Acknowledgments

The authors would like to thank Greg Iles (Imperial College, London) and Andrew Rose (Imperial College, London) for providing the original sorting design and their appreciated input for this paper.

References

[1] H. Yasuura, N. Takagi and S. Yajima, The parallel enumeration sorting scheme for VLSI, IEEE T. Comput. C-31 (1982) 1192.
[2] M. Fingeroff, High-level synthesis blue book, Xlibiris, U.S.A. (2010).
[3] G. Martin nd G. Smith, High level synthesis: past, present, and future, IEEE Des. Test. Comput. 26 (2009) 18.
[4] D. Gajski, T. Austin and S. Svoboda, What input-language is the best choice for High Level Synthesis (HLS), in the proceedings of the 47th Design Automation Conference (DAC10), June 13–18, Anaheim, U.S.A. (2010), p. 857.
[5] P. Coussy and A. Morawiec, High-level synthesis: from algorithm to digital circuit, Springer, U.S.A. (2008).