Xilinx Vivado High Level Synthesis: Case studies

Declan O'Loughlin[1], Aedan Coffey[1], Frank Callaly[1], Darren Lyons, Fearghal Morgan[1]

[1]*Department of Electronic & Computer Engineering*
NUI Galway, Galway, Ireland
Email: fearghal.morgan@nuigalway.ie

Abstract—**This paper presents case studies on the application of the Xilinx Vivado High Level Synthesis (HLS) tool-suite for C++-based design capture, simulation and synthesis to Hardware Description Language (HDL) format, and further to FPGA hardware implementation. HLS reduces the effort of HDL design capture and debug while allowing flexibility in the final hardware implementation in order to meet design constraints.**

HLS is not yet widely used. This paper demonstrates the practical steps in using HLS and the resulting hardware implementation. Case studies illustrate the effectiveness of HLS as a developing efficient and flexible design capture to FPGA implementation approach.

The paper presents four HLS design examples, including a multiplexer, counter, register block and a skin detection image processing algorithm. Xilinx PlanAhead EDA tool-suite is used to generate a Xilinx Spartan-6 FPGA bitstream from the Xilinx Vivado HLS-synthesised HDL model. Each design has been implemented and tested in FPGA hardware using the Vicilogic automation and proto-typing tools developed by the authors. These tools automate the integration of designs with an FPGA IP core, which supports Ethernet I/O, SDRAM interface and a register-based I/O system. The Vicilogic Python client application environment enables GUI-based development and testing of the hardware implementation.

Keywords **– High-level synthesis, HLS, FPGA, EDA tools**

I INTRODUCTION

Hardware Description Languages (HDLs) are the more recent traditional digital systems design entry method. This paper presents case studies on the application of the Xilinx Vivado High Level Synthesis (HLS) tool-suite for C++-based design capture, simulation and logic synthesis to Hardware Description Language (HDL) format. HLS reduces the effort of RTL HDL design capture and debug. HLS also allows flexibility in the final hardware implementation in order to meet design constraints.

High-level synthesis takes a complete behavioural C++ description of a system along with a series of directives that describe the architectural constraints, and automatically generates a HDL design description.

Wakabayashi [1] indicates that digital logic designs of the order of one million gates require three hundred thousand lines of RTL code. This is not only costly and time-consuming to develop, but also difficult to debug and verify.

This paper presents four HLS design examples, including a multiplexer, counter, register block and a skin detection image processing algorithm. Xilinx PlanAhead EDA tool-suite is used to generate a Xilinx Spartan-6 FPGA bitstream from the Xilinx Vivado HLS-synthesised HDL model. The paper illustrates the effectiveness of HLS as a developing approach for efficient and flexible design capture to FPGA implementation. HLS is effective in developing and testing early design ideas prior to full HDL generation and RTL simulation.

Each design has been implemented and tested in FPGA hardware using the Vicilogic automation and proto-typing tools developed by the authors. These tools automate the integration of designs with an FPGA IP core, which supports Ethernet I/O, SDRAM interface and a register-based I/O system. The Vicilogic Python client application environment enables GUI-based development and testing of the hardware implementation.

Although HLS is not yet widely used, some industrial design examples are described in [2]. This paper demonstrates the practical steps in using HLS,

and the resulting hardware implementation details for a number of design examples.

HLS is a flexible design capture method. A HLS model of a system can be synthesised into many different HDL models, each meeting different timing, clock and area constraints. Salminen [3] presents surveys indicating that up to 50% of development time is spent in verification, with this percentage increasing as designs increase in complexity. HLS offers opportunities to reduce verification time. HLS allows the creation of abstract C++-based functional models, which can be readily tested and verified, prior to generation of HDL design descriptions. This can reduce the testing and debug effort on the resulting HDL models. In contrast, directly captured HDL models require significantly more testing.

FPGAs are increasingly being used in parallel, high-performance computing applications Hardware/Software architectures are becoming increasing common. HLS lends itself to this approach, enabling partitioning of C-level functions between processor(s) and HLS-synthesised FPGA hardware blocks. This facilitates easier design by non-traditional digital design experts.

The paper introduces the principles of HLS, the available HLS tools, and the HLS-based capture to design process. The examples illustrate the issues and capabilities of Xilinx Vivado HLS. The HLS-synthesised HDL is used with the Xilinx PlanAhead EDA tool-suite to target Xilinx Spartan-6 FPGA implementation.

The structure of the paper is as follows: Section II introduces the principles and application of HLS. Section III illustrates the use of Vivado HLS for basic combinational and sequential examples, namely multiplexer, counter and register designs. Section IV presents the HLS-based process applied to a skin detection image processing algorithm [4]. Section V illustrates the Vicilogic ethernet-connected client and python-based API. Section VI concludes the paper and proposes future work.

II HLS Tools, Principles and Process

This section introduces the principles and application of HLS. Over the years, the FPGA design community has moved through many design abstraction layers to facilitate increasingly more complicated designs. Netlists, which map a design to a series of interconnected instances from a pre-built library reduce the layout effort. Register Transfer Level (RTL) descriptions help to define a functionally correct model from which the netlist can be automatically generated. A more recent development is the use of digital functional models described using a higher-level language such as C++, and its use to quite automatically generate RTL models.

Vivado HLS is the Xilinx HLS engine, accepting descriptions in C, C++ or SystemC and generating VHDL, Verilog and SystemC RTL descriptions. These descriptions can be packaged as IP blocks and imported directly into the Vivado Design Suite for series 7 Xilinx device implementation. The RTL models can also be used with Xilinx PlanAhead or ISE EDA tool-suites to target series 6 (or earlier) FPGA device implementation.

High-level synthesis follows two distinct phases, namely

1. algorithm synthesis, in which the behaviour of the system is extracted and scheduled over a number of clock cycles
2. interface synthesis, where the variables and parameters of the top-level function are transformed into appropriate ports with appropriate hand-shaking signals.

Vivado HLS achieves task 1 in two phases as follows. Firstly, the control and data paths are extracted from the HLS model by unrolling the loops and examining the model conditional statements. Next, in the scheduling and binding phase, an optimum implementation is found using knowledge of the clock, the latency of the hardware, and the user constraints. For example, four multiplications can be achieved in many different ways, namely:

1. four clock cycles with one adder and one multiplier
2. two clock cycles with two adders and two multipliers
3. one clock cycle with three adders and four multipliers

Option one is the slowest, but also uses the least amount of hardware. Conversely, option three is the fastest, but also uses the most hardware and is only viable if the multiplication and addition can be completed within one clock cycle [5].

Task 2, interface synthesis, is achieved by analysing the parameters of the top-level function and determining their role. For example, a pass-by-value primitive data-type (e.g. an *int*) is be synthesised into an RTL input with or without valid and acknowledgement signals as directed. Pass-by-reference values (e.g. pointers) must be further analysed to determine if they are inputs, outputs or used as both inputs and outputs, and synthesised accordingly.

Figure 1 illustrates the typical HLS design flow. Once a specification has been determined, this is expressed in an executable C++ model which is then converted into an RTL model using Vivado HLS. This would traditionally be an iterative process of architectural specification, RTL coding and RTL debugging. RTL can still be verified using traditional methods such as simulation. However, the detection of design issues and bugs can be more efficient using C, C++, or SystemC verification [6]. Vivado HLS also offers a co-simulation feature that can be used to help verify the generated RTL designs. Using any simulator, the C/C++/SystemC test-bench can be used to generate appropriate test vectors for the generated

RTL models (taking the generated interfaces into account). This can help provide an initial indication that the generated model is functionally correct subject to further verification and timing analysis.

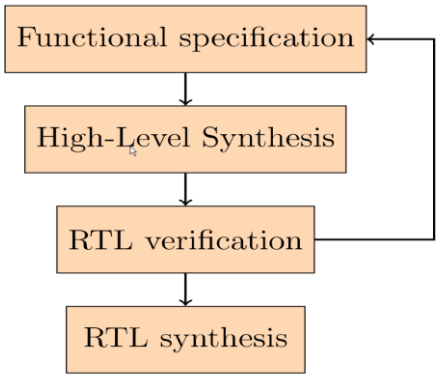

Figure 1: HLS Design Flow

III VIVADO HLS

This section illustrates the use of Vivado HLS for basic combinational and sequential examples. The section also illustrates the Vicilogic user design wrapping process [7] for creation of FPGA bitstream.

Firstly, a C++ functional design is verified with a C++ test-bench. The C++ design is then synthesised to an RTL HDL-based model. Using co-simulation, the HDL is stimulated using test-vectors generated by the original C++ level test-bench which aids functional verification.

The RTL model is then imported into Xilinx PlanAhead and RTL simulation is performed. Once the behaviour of the system has been verified, the Vicilogic user design wrapper is used to merge the user design with the Vicilogic IP core [7]. The design targets the Digilent Nexys3 Xilinx Spartan-6 FPGA. The IP core also includes a controller for the Nexys3 SDRAM, BRAM and display controller interface, and user clock control. The wrapper provides a register interface through which the Vicilogic client application can control user design inputs and read any signals within the user design in real time. The wrapper automatically generates the Spartan-6 FPGA configuration bitstream.

Each of the design examples (multiplexer, counter, register block and skin detection image processor) implemented using the HLS to FPGA implementation process are described below.

a) Multiplexer Example

Figure 2 illustrates a C++ description of a 4-to-1 multiplexer (4-bit data). This design demonstrates four important factors, described below.

Firstly, the use of arbitrary precision integers to model bus widths. Data-types *data_*t and *sel_*t are defined as four and two bit unsigned integers respectively, and are synthesised to four and two bit wide buses respectively. These data-types are defined in a header file included with Vivado HLS called *ap_int.h.*

Secondly, this design illustrates the use of standard C++ constructs to model functional behaviour. This is a purely combinational multiplexer design and the top-level function represents the complete behaviour of the combinational design. A case statement is used, similar to a case statement in a HDL model. *sel_t* is an unsigned two-bit integer.

Thirdly, the input and output parameters are shown in the function definition. The inputs in this case are all primitive types and are passed-by-value as integers. The output is a pointer to an unsigned integer. As this pointer is only written, the HLS engine correctly determines that this is an output port and synthesises the port accordingly.

Fourthly, two types of directive are shown. Firstly, an application control directive that indicates to the HLS engine that no hand-shaking at a block level should be used, i.e, no start, stop, done, ready and idle signals. The second directive indicates that for the output port, *muxOut,* no valid or acknowledgement signals should be used. The combined effect of both of these directives is a purely combinational design with the five specified inputs and one output.

```
1  #include "mux.h"
2  #include <ap_int.h> // Arbitrary width integers
3  typedef ap_uint<2> sel_t;
4  typedef ap_uint<4> data_t;
5  void mux(sel_t sel,
6      data_t muxIn0, data_t muxIn1,
7      data_t muxIn2, data_t muxIn3,
8      data_t *muxOut) {
9  #pragma HLS INTERFACE ap_ctrl_none port=return
10 #pragma HLS INTERFACE ap_none port=muxOut
11   switch (sel) {
12     case 0: *muxOut = muxIn0; break;
13     case 1: *muxOut = muxIn1; break;
14     case 2: *muxOut = muxIn2; break;
15     case 3: *muxOut = muxIn3; break;
16     default: *muxOut = muxIn0; break;
17   }
18 }
```

Figure 2: C++ Four-to-one four-bit multiplexer.

b) Four-bit Counter

Figure 3 illustrates a complete C++ description of a 4-bit counter with an enable signal. This design demonstrates a number of interesting features. Although this is a sequential design, there no clock or reset signals are included in the C++ function definition. These are inserted by the algorithmic phase of the high-level synthesis process. However the enable signal, which is used to control whether or not the count variable is incremented on each clock edge (execution of the function), is included in the functional definition.

This design also illustrates the use of registers. *count_value* is a global variable in the C++ model that represents the current value of the counter. The HLS engine determines the most appropriate

representation for this variable. Typically single variables, as in this case, are synthesised to registers, i.e, internal signals in the system RTL architecture. This signal is incremented on each active clock edge if enable is asserted.

Since *count_value* is an unsigned four-bit integer, not only will it be synthesised to a four-bit wide bus, but the overflow will work as expected in C and VHDL, i.e, when at the maximum value, 2^n-1, 15 it will loop to value 0 and any carry will be ignored. Care has to be taken with the representation of the numbers; a signed representation will behave substantially differently incrementing from 2^{n-1} to minus 2^n.

```
#include "counter.h"
typedef ap_uint<4> counter_t;
counter_t count_value=0;
void counter(bool en, counter_t *count) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=count
  if (en)
    *count = count_value++;
  else
    *count = count_value;
}
```

Figure 3: C++ Four-bit counter with enable.

IV IMAGE PROCESSING SYSTEM

Figure 4 shows the block diagram of a skin detection image processing system. The Vicilogic client application captures video frames, transfers these via ethernet packets to the FPGA Vicilogic [7] core and into SDRAM on the Digilent Nexys3 Spartan-6 FPGA board. An image processing function implemented on the FPGA reads each word of the SDRAM image data in turn, performs a skin detection algorithm and writes the result to SDRAM. An 8-byte read/write control register block within the FPGA enables parameterisation of the image (function index, memory start address, image size). Other supported image processing functions include frame subtraction and image warping.

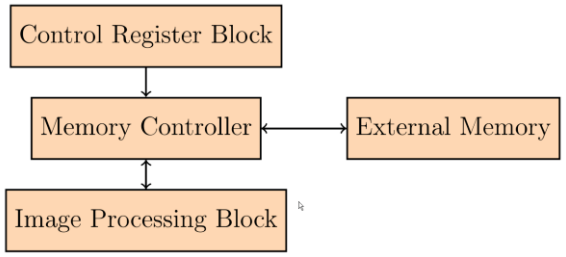

Figure 4: Image processing system.

a) Control Register Block

Figure 5 shows the C++ description of the 8-byte control register block. In many respects, this design is similar to the counter. It consists of a global variable, a number of inputs and one output. However, in this case, the global variable, *registers*, is an array of bytes. Vivado HLS correctly infer that the registers should be stored in memory and hence the registers are implemented as 8 bytes of BRAM.

```
#include "ctrlReg.h"
data_t registers[8];
void ctrlReg(data_t dataIn, addr_t addr, rw_t rw,
    data_t *dataOut) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=dataOut
  if (rw == 0) // read if high, write if low
    registers[addr] = dataIn;
  else
    *dataOut = registers[addr];
}
```

Figure 5: C++ 8-byte R/W Control Register Block.

Table 1 shows the resource utilisation report for the control register block, obtained from the synthesis stage of Xilinx PlanAhead. It can be seen that HLS minimises the use of registers and uses one BRAM. The synthesised model of the VHDL model of the control register block uses many more registers and LUTs but no BRAM. The resource directive can be used in the HLS model to specify what resource should be used to implement any variable; in this case the type of memory to be used.

Resource	HLS	VHDL model
Registers	9	64
LUTs	10	24
BRAM	1	0

Table 1: Control Register Block synthesis utilisation report

b) Image Processing Block

Figure 6 illustrates an extract of the skin detection image-processing algorithm. Detected skin coloured pixels are marked in the result image. Others are cleared (image RGB pixel values =0).

```
  case 4:
    if (testA(dDatFromRam) && testB(dDatFromRam) && testC(dDatFromRam))
  *dspDat2Ram = pixel(0xff, 0, 0); // Found
    else
  *dspDat2Ram = pixel(0xff, 0xff, 0xff); // not found
    break;
```

Figure 6: Extract from skin-detection algorithm

Depending on the clock speed and design constraints, the HLS engine determines whether or not the algorithm can be completed in one clock cycle and the appropriate HDL design is generated. In this case, this algorithm can complete in one clock cycle at a clock

frequency of 50 MHz or below. At a clock frequency of 100 MHz, this requires two clock cycles.

Table 2 shows the utilisation report obtained from the synthesis stage of Xilinx PlanAhead. The resource usage for the HLS generated RTL model is slightly more efficient than that for the VHDL model,

Resource	HLS	VHDL model
LUTs	148	156

Table 2 : Image Processing Block synthesis utilisation report

Figure 7 shows an example of an image following the process of FPGA-based skin detection. The deep red represents the facial area.

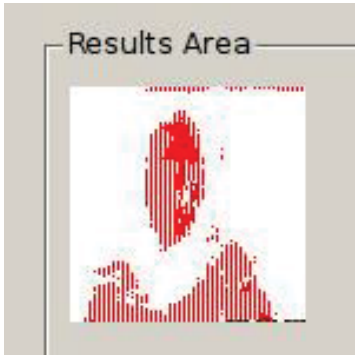

Figure 7: Example face detection

V HOST-CLIENT COMMUNICATIONS

The Python host reads an image in real-time from a web camera or pre-defined images from files, transmits the image to FPGA module SDRAM (over the Ethernet) [7]. The python GUI provide real time image processing results.

VI CONCLUSIONS

As FPGAs grow in power, complexity and include additional processing cores as well as reconfigurable logic, HLS has the opportunity to lower the barrier to entry for acceleration of certain functions and algorithms in reconfigurable logic. Existing algorithms can be quickly and accurately implemented in RTL without having to manually design and capture the architecture in a HDL. HLS offers reduced development time. The generated RTL models are often at least as efficient as those obtained from synthesis of hand-coded RTL.

This paper has presented case studies on the application of the Xilinx Vivado High Level Synthesis (HLS) tool-suite for C++-based design capture, simulation and synthesis to Hardware Description Language (HDL) format, and further to FPGA hardware implementation. The paper demonstrates the practical steps in using HLS and the resulting hardware implementation. The Vicilogic Python client application environment enables GUI-based development and testing of the hardware implementation. Case studies illustrate the effectiveness of HLS as a developing efficient and flexible design capture to FPGA implementation approach.

REFERENCES

[1] K. Wakabayashi, *"C-based behavioural synthesis and verification on industrial design examples"*, in Proc. ASPDAC '04 pp. 344-348.

[2] Cony, J. et al., *"High-Level Synthesis for FPGAs: From Prototyping to Deployment"*, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 30, no. 4, pp. 473-491, Apr. 2011.

[3] Salminen, E. *"Efficient IP Reuse for FPGAs"* in TehoFPGA, Tampere, 2013.

[4] Kovac, J., Peer, P., Solina, F. *"Human skin color clustering for face detection"*. EUROCON Computer as a Tool, 2, Sept. 2003.

[5] *"Vivado Design Suite User Guide: High-Level Synthesis"*, UG902 (v2012.2), Xilinx, San Jose, CA, 2012.

[6] Fingeroff, M. *"High-Level Synthesis: The Blue Book"*, Oregon: Mentor Graphics, 2010.

[7] Morgan et al.; "Vicilogic: Digital Logic Application Prototyping and Online Education", submitted to FPL 2014 (under review).