cadence®

How the Productivity Advantages of High-Level Synthesis Can Improve IP Design, Verification, and Reuse

By Dave Pursley, Cadence Design Systems

Engineering teams are under more pressure than ever before—systems on chip (SoCs) are growing more complex and design schedules are increasingly tighter. With its productivity advantages, high-level synthesis (HLS) has long been touted as part of the solution, but its sweet spot has traditionally been limited to datapath-centric blocks. Moreover, design productivity is only one part of the equation. Verification is often an even bigger hurdle.

This paper discusses how HLS can be used to improve the design, verification, and reuse of intellectual property (IP). The paper also introduces a new HLS tool that provides excellent power, performance, and area (PPA) results across the digital design space.

Contents

Introduction

As SoCs have grown more complex, so, too, have the design and verification tasks. To help manage this, engineering teams are experiencing great efficiency by starting their tasks with a high-level C++ model to verify functionality. They then use the SystemC C++ class library to evaluate architectural tradeoffs, while verifying that those architectures are still functionally correct. HLS synthesizes the architecture to register-transfer level (RTL), ensuring that the design intent of the architectural model is actually reflected in the hardware. Moreover, the same environment used to verify the architecture is reused to verify the RTL.

In this paper, you'll learn more about how HLS can be used to streamline the design and verification process, while also generating better PPA and a more efficient IP development and reuse process. The paper includes case studies featuring the use of HLS for an L2 cache subsystem and a configurable processor for baseband, PHY, and video applications—designs that would not typically be thought of as targets for HLS.

Describing Hardware with C++ and SystemC

Given the complexity of today's SoCs, a model that closely reflects the intent of the hardware architecture is the most feasible way to both design and verify such chips. Object-oriented capabilities in C++, such as templates, classes, polymorphism, and operator overloading, can be used to manage design complexity in hardware in the same way as in software.

For example, when reading a memory, your source code can be as simple as follows.

```
int address, a, memory[1024];
…
a = memory[address];
```

With operator overloading, the memory read function can be invoked using an array access notation as shown above. The details of the hardware access to a memory are abstracted, making the code compact and readable. Those details are instead implemented in a C++ class that uses SystemC to set the chip enable and read/write signals and to drive the address signals, adhere to setup and hold times, etc.

Other interfaces can be similarly abstracted. For example, reading from an input can be as simple as a C++ assignment statement:

```
int val;
…
val = port0;
```

Again, a C++ class encapsulates and implements all of the details of the communication. The actual hardware communication could be a simple port read, a handshake, a FIFO, or a bus interface. It could even communicate across clock domains, requiring a clock domain crossing (CDC). These different types of classes essentially become IP for use in your C++ or SystemC designs and testbenches. They are available in the new Cadence® Stratus™ High-Level Synthesis (HLS) platform.

Case Study #1: L2 Cache Subsystem

To illustrate our discussion, let's take a look at how a small engineering team turned to SystemC when tasked with developing and verifying the L2 cache subsystem of an advanced multi-core, multi-threaded processor. The subsystem, shown in Figure 1, included the required interfaces to a local high-speed external bus and L1 cache. The local high-speed external bus needed to connect to various external interfaces, including DDR3, Flash, Ethernet, and DVI.

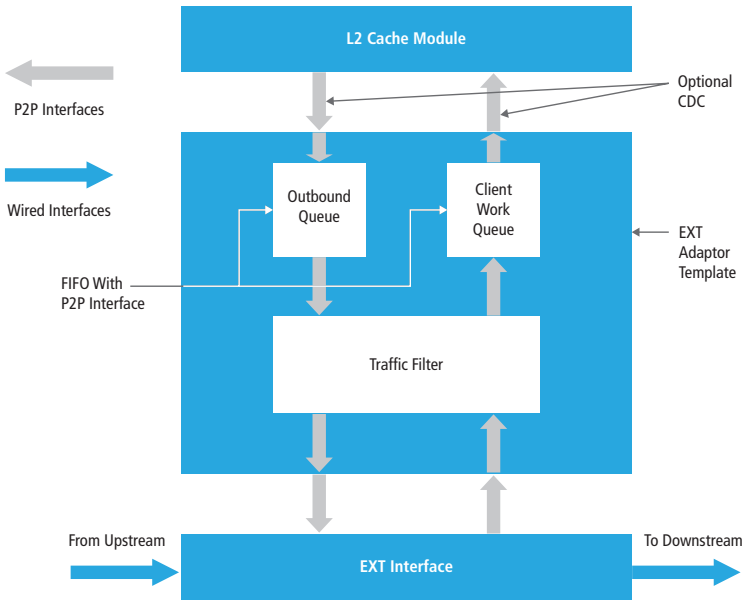

Figure 1: L2 cache interface

The team coded the source for this chip entirely in SystemC. The engineers then were able to use the resulting model as a behavioral simulation model for extensive design verification, as well as for HLS using a commercial tool to create Verilog RTL. As a proof of concept, the team moved the generated RTL code through a typical RTL design flow and implemented it on an FPGA.

For this L2 cache subsystem, the engineering team needed to develop many interfaces between off-chip resources and the external interface (EXT) local high-speed bus. The L1 and L2 caches also had to be built with interfaces to the EXT. And, the entire cache model had to be built in SystemC. The team took advantage of pre-verified interface IP available in the Stratus HLS tool. The IP encapsulates the detailed signal-level protocols, providing a simple and consistent API for all communication, regardless of the actual interconnect topology.

The Stratus HLS flexible point-to-point (P2P) interface IP proved especially beneficial for this team. Its simple put() and get() API for writing and reading hides the complexity of a handshaking interface that supports streaming or non-streaming data transfer at arbitrary throughput, up to one data value per cycle. It ensures reliable transfer by allowing the downstream module to assert its busy output and stall the upstream module. As a result, data won't be lost in cases when the downstream data consumers can't keep up with the data providers.

By using the pre-verified interfaces and memory models, the engineering team estimates that it reduced its design time from two years down to five months.

Case Study #2: Using HLS for Architectural Evaluation of a Configurable Accelerator

In our other use case, let's take a look at a low-power configurable accelerator developed for baseband, PHY, and video applications (Figure 2). The accelerator fits into an LLVM compiler tool chain. Here, the engineering team used HLS to conduct its evaluation of potential architectures written in SystemC. Ultimately, the team wanted to determine whether a programmable or a hard-coded accelerator would best meet its needs. To find its ideal architecture, the team experimented by:

- Varying the number of accelerators to match the amount of parallelism extracted from the algorithms
- Varying the memory interface
- Varying the instruction set to include advanced autonomous digital signal processing (DSP) instructions

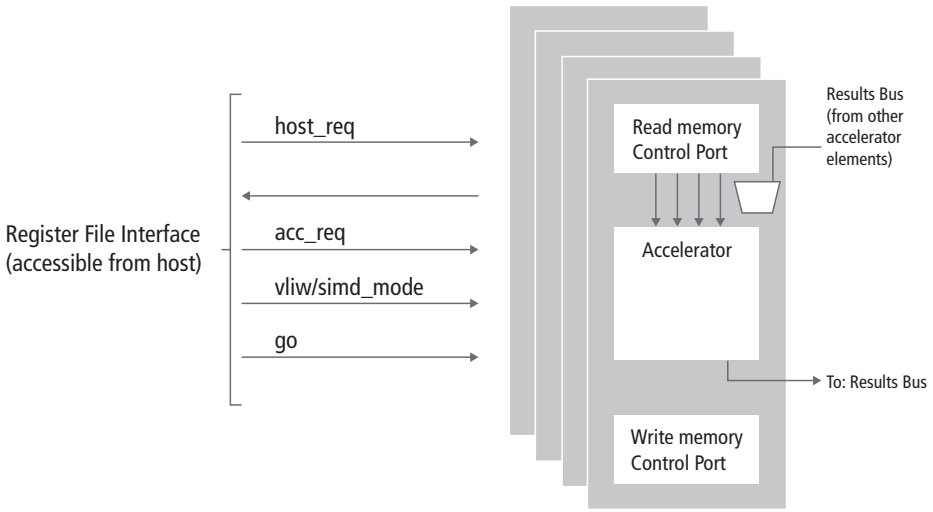

Figure 2: Base architecture for evaluation of programmable vs. hard-coded accelerators

For its assessment, the engineers first compiled an application to assembler using a configurable LLVM compiler tool chain. They ran the compiled program on the SystemC model of the accelerator for functional verification. Then, the team used an HLS tool to create an RTL implementation, running the same compiled program on the RTL. Through the RTL simulations, the engineers got accurate estimates of PPA with its existing RTL design tools.

After running a series of benchmarks, the team learned that, typically, two to four instructions would be executed in parallel per basic block in the C code. They measured architectural performance by running common DSP and baseband algorithms such as Viterbi and turbo decoders. The evaluation steps, detailed as follows, resulted in very accurate switching characteristics for performing power analysis:

1. Design an instruction set to suit individual applications
2. Compile the application into assembly code, and run this code against the SystemC model by the test harness
3. Create a Verilog RTL model of the hardware with the HLS tool and run the same assembly code against that Verilog model

Overall, the team found that SystemC and HLS provided an efficient methodology for evaluating architectural decisions. Some of the tradeoffs they found were surprising, highlighting the importance of being able to create and verify RTL directly from their architectural models. For example, they found that adding support for four complex DSP instructions to the accelerator required a surprisingly small additional silicon area, because HLS was able to find a schedule that effectively shared the functional units.

The team was concerned about coding efficiency, namely whether they could write code for high-level synthesis in natural C coding style and still achieve good synthesis results. For the programmable accelerator, the team found it easy to code the instruction dispatch model as a simple switch statement. Each case then represented one of the op-codes for the accelerator. Such a coding style is natural and extensible. The engineers found that if they decided to add an op-code, they just needed to add another case to the switch. The Stratus HLS tool then handled all of the details of the modified state machine.

Essentially, the Stratus HLS tool synthesized RTL from the team's SystemC source and provided integrations for running simulation and all of the RTL synthesis, simulation, and power analysis tools. The team found the key benefit to be the tool's ability to automate the data transfer between all of the individual tools. Since it already provided default scripts for running other tools, the team found it easy to create a working design flow. Also, as the engineers were developing source code, they continually ran SystemC synthesis. Once the source code was complete, they were able to quickly generate the RTL needed to complete their analysis.

Fast Path to Production Silicon

The two use cases discussed in this paper demonstrate how HLS technology like Cadence's Stratus platform can support a fast path to production silicon. The Stratus HLS tool allows for implementation-independent SystemC models, as well as fast design derivatives for varying requirements from a single, high-level model. The platform is easily re-targetable and supports IP reuse and engineering change orders (ECOs). With early access to fast TLM simulation models, a consistent verification platform from TLM to the netlist, and a direct link to the Cadence verification flow, users gain a productive verification platform. The Stratus platform also delivers high quality of results (QoR) via optimized PPA tradeoffs and a direct link to the Cadence implementation flow.

Summary

HLS has long been promoted as a technique to increase design productivity, especially on datapath-centric blocks. In this paper, via use cases not typically associated with HLS, we showed how HLS can be used to improve design, verification, and reuse of IP. This paper also introduced Cadence's Stratus HLS tool, which provides a productive design and verification platform with high QoR across the digital design space.

For Further Information

Learn more about the Cadence Stratus HLS platform here: http://www.cadence.com/products/sd/stratus/pages/default.aspx.

Sources

Tessier, Thomas and Dr. Hai Lin, Daniel Ringoen, Eileen Hickey, and Steven Anderson. "Designing, Verifying and Building an Advanced L2 Cache Sub-System Using SystemC." Paper presented at DVCon, March 2012. http://events.dvcon.org/2012/proceedings/papers/03_3.pdf

Fox, Andy and Tigran Sargsyan and Steven Anderson. "Architectural Evaluation of a Programmable Accelerator for Baseband, PHY, and Video Applications Using High-Level Synthesis." Paper presented at DVCon, March 2013.