High-Level Modeling and Hardware Implementation with General-Purpose Languages and High-level Synthesis

John Sanguinetti and David Pursley
Forte Design Systems

Abstract

This paper addresses high-level hardware modeling and top-down design methodology. While it is common practice to begin the design process with a system model written in a general-purpose programming language, in order to proceed in a top-down fashion from that initial model, two technologies are required. Modeling capabilities in a general-purpose language must be made available, so that lower levels of abstraction can be represented in the GPL. And, high-level synthesis that can produce an acceptable implementation from a high-level description is required. Both of these technologies are nearing maturity, which will enable a new era of system design.

Introduction

This paper will address high-level hardware modeling and top-down design methodology. We take the position that designing complicated systems in a top-down manner is a proven methodology, as demonstrated by decades of experience designing systems from aircraft carriers to operating systems. As hardware systems have become more complex, there has been a natural progression to designing them using this methodology, but a number of impediments have not yet been overcome.

The ideal methodology would be to write an executable description of the system (a model) which could be used to explore the design space. Then, once the appropriate design had been settled on, push a button and automatically generate an implementation which meets the design constraints. A key aspect of this methodology is that the specification, or model, be rendered only once. That is, after the model has been written, any translation into another language or representation should be done automatically.

Another observation that we can make is that general-purpose programming languages (GPLs) are nearly universally used for modeling hardware/software systems because they have been found to be most convenient for representing algorithms. The single greatest source of reusable IP is algorithms written in C. If we are to reach this ideal state, the system model must be derived from code written in a GPL.

The current state of the art is not yet at this ideal. While we have the tools to model complex systems, we do not have the means to automatically generate an acceptable implementation from a high-level model. It is our contention that two technologies are required to solve this problem:

1. additions to general purpose programming languages that facilitate system modeling, like SystemC,
2. high-level synthesis using the augmented GPL as input and producing output which is suitable as input to the standard hardware implementation tool flow.

These two technologies are reaching a level of maturity where high-level models can be used directly for hardware implementation.

The problem

Top-down, iterative design starts with a system model expressed in C++ and ends with a hardware description expressed in GDS-II. There is a well-defined set of tools which can transform a Register-Transfer Level description in a hardware description language (e.g. Verilog) into GDS-II. The gap between a model expressed in C++ and an RTL model expressed in Verilog is the problem.

Modeling vs. Design

The first representation of a design is called a model, though it could also be called a specification. While models can be produced in a variety of forms, we are concerned here with executable models, which are typically just programs. Such a program captures the intended behavior of the system at some level of abstraction. That is, it produces the desired outputs to a range of inputs. Typically, the initial model does not capture detailed behavior, except to the extent required for the initial design decisions. For example, a fundamental algorithm like an MPEG decoder might be included, but the processor to memory bandwidth might not be included.

The reason we call the initial design a model is that it is typically the vehicle for design exploration. The designer may try several different variations of the model. For example, he may try out different implementations of an algorithm, or different combinations of resources. It is at this experimentation stage that the most fundamental decisions about the system are made. Things like how many processors, which algorithms will be used, which ones go in hardware and which in software, characteristics of the memory architecture, data connections, all are fundamental decisions which will shape the eventual implementation of the target design, and which may be made based on information gained from experimentation with the initial model. For this reason, the initial model is often called an architectural model.

It is a short conceptual step from architectural model to system implementation. After all, the model is executable, and the target system is executable, so it is just a matter of transforming one into the other. The rub, of course, is in the details. The transformation must conform to the design constraints, which may not be captured in the architectural model. As the constraints are added, as well as the design features those constraints imply, the model becomes a more precise specification and, ultimately, a complete representation of the design. This is the very essence of top-down design.

As a practical matter, it is easiest if, in this design process, small steps can be made without large discontinuities. In the current state of the design art, there is a large discontinuity somewhere between the architectural model and the complete design representation, due to the simple fact that the downstream tool flow requires the design representation to be in an HDL at a register transfer level, but the architectural model is written in a GPL at a behavioral level. This discontinuity must be overcome before we can expect wide adoption of top-down, iterative design starting from a high-level model.

The most significant consequence of the language discontinuity is that once the model has been changed into an RTL/HDL representation, many design characteristics are effectively frozen, and cannot be changed without a significant amount of effort. That is, in order to change significant design characteristics, a new translation from higher-level GPL to RTL/HDL is required. This process is so costly in terms of elapsed time that, in most cases, it cannot be done and still meet project deadlines.

There are two technologies needed to bridge the discontinuity:

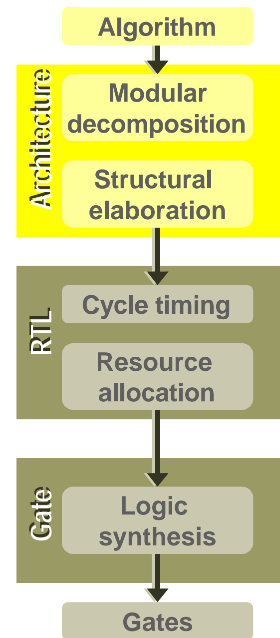

Figure 1. Top-down Design Flow

GPLs must be augmented to facilitate both system modeling and hardware description

C++ and Java are two GPLs which have an object orientation that allows these required features to be added in a standard manner. SystemC [1] is a class library which augments C++ to add both system modeling features and hardware modeling features, down to and including the register-transfer level. SystemC's hierarchy of classes provides a framework for building up a very general set of layers of abstraction from RTL all the way to application-specific layers.

High-level synthesis must use the augmented GPL as input and produce HDL output suitable for downstream tools

High-level synthesis provides the connection between the system model and the implementation. It is a necessary technology to enable the wide-spread use of high-level modeling and top-down design for hardware systems, because without it, there is the extra work of both recreating the implementation representation, and also verifying that it conforms to the system model.

The two technologies are both necessary. There is little incentive to use SystemC in the absence of high-level synthesis, and there is also little incentive to use high-

level synthesis without the ability to do hardware modeling in a GPL.

SystemC

SystemC implements several levels of abstraction. There are currently 3 broad categories of classes, which make up levels of abstraction:

1. RTL
   The RTL classes make up the bulk of SystemC, and implement a modular structure, process concurrency, and bit-accurate data types. This layer implements all of the features that are normally found in the RTL subset of an HDL (e.g. Verilog).
2. Communication
   The communication classes implement data transmission and synchronization protocols which are built on top of the basic process concurrency controls of the RTL layer. These classes are typified by channels which pass data through module ports with a handshake.
3. Verification
   Verification classes make up a set of abstractions which are used for creating test benches. A test bench is really just a (more or less) abstract model of the design under test's environment. Verification classes provide features for randomization, data logging, and protocol matching.

Using the verification classes and the communication classes, a test bench can be created which works for the design under test at all levels of abstraction. In effect, the test bench is the model of the "rest of the system", which might very well include the software that will run on it. Having a constant test bench is crucial for design exploration. If the test bench has to change when some change is made to the design under test, then it is difficult to conduct reliable experiments.

We should note that SystemC is an open source C++ class library and is, in effect, a reference implementation of a standard set of classes. We can expect to see other implementations of SystemC, the first of which is Extended SystemC. Extended SystemC is a newer implementation of the SystemC classes which has better performance and fewer restrictions than the reference version. The examples discussed later used Extended SystemC.

High-Level Synthesis

High-level synthesis, sometimes known as behavioral synthesis, has been around for a long time and it hasn't been widely adopted. The main reason that it has not proven all that useful is that it started at the wrong point. Without the link with a GPL, there was no natural starting point. Architectural models were still written in C++ and a translation to an HDL was still required. SystemC and CynthHL change that.

CynthHL [2] is one of a new generation of high-level synthesis products that takes C++/SystemC as input and produces Verilog/RTL as output. This output can be fed directly into a standard logic synthesis product. In fact, in the very near future, logic synthesis products will accept SystemC/RTL as input directly.

Typically, high-level synthesis performs two functions: scheduling and resource allocation. That is, a computation is written as an algorithm encapsulated in a module and the module's communication protocol is represented either in a communication class or directly as a signal-level handshake. Then, the synthesis program creates a state machine and data path that implements the computation. The synthesizer must select the necessary set of functional units for the data path and schedule them appropriately using control logic, satisfying area constraints (how many functional units) and latency constraints (how many cycles the computation takes).

Design Exploration

The key benefit of using a top-down design methodology is that design decisions can be made at the appropriate time, which is another way of saying that they can be changed with a minimal amount of wasted effort. Each time a decision is made, it can be tested in the context of the overall system model to see its effects. This, of course, is the whole point of using system models in the first place. A common problem, however, is that the implications of some design decisions, particularly as they affect latency and area, cannot be determined until the design has reached the RTL or netlist stage. Without the availability of high-level synthesis, backing up after the translation to RTL has been made may be difficult or impractical.

A lower level of design exploration involves trying different RTL implementations of a single architectural model. Decisions at this level would include things like how many functional units to use in the data path (adders, multipliers, etc.) or how much parallelism to use. Being able to easily generate multiple RTL implementations from an architectural model is a clear benefit of high-level synthesis, although previous behavioral synthesis products have been unable to deliver on this promise. More precisely, they were able to create an RTL implementation from an architecture, but the verification of each implementation was very

difficult if not impossible, due to the lack of a GPL environment.

Using high-level synthesis, practical implications of design decisions can be determined at the time they are needed, allowing better decisions to be made. This improved ability to explore the design space is the key benefit of high-level synthesis, and the top-down design methodology it enables.

An AES encryption algorithm

This is an example of design exploration of an AES encryption algorithm using Extended SystemC for modeling and CynthHL for high-level synthesis. Area was the top design goal, with performance a secondary concern ("make it go as fast as possible in minimum area"). By applying different constraints and directives with CynthHL, the designer was able to create many different RTL implementations of the same design. Each design could be verified with the original testbench, guaranteeing that the synthesis results were not only gates, but "correct gates."

An automated flow for design exploration was used, allowing trade-offs between competing design goals (area vs. performance in this case) to be made using actual synthesis numbers. The procedure was:

1. unwind all the loops completely
2. iteratively decrease the latency constraint

It turns out that with each iteration, the area change was fairly minimal, so with the fastest latency constraint that could be met, overall area had only increased by ~20%. Unwinding loops completely allowed a lot of constant propagation, which immediately removed a lot of computation. However, it also increased the number of states in the FSM considerably. As the latency constraint tightened, the relatively small functional units began to be placed in parallel (increasing area), but that was partially offset by the decrease in FSM size (fewer cycles = fewer states). The net result was a lot of speed-up at only moderate area cost.

This was an interesting, unexpected trade-off, increasing performance by 5x while increasing area by only 20% over a reference design that had been done by hand. Had the designer known about this trade-off, he would have chosen this design point over the slightly smaller but much slower reference design. However, there is little chance the designer would have stumbled on this design point with a standard design process.

An image compression algorithm

The design of an image compression algorithm illustrates the value of architectural exploration. Using Extended SystemC and CynthHL, the designer was able to meet aggressive performance constraints and verify that the design decisions did in fact produce a correctly functioning system in the face of an apparently non-viable starting system architecture.

The design was required to process an image at a rate of 66 frames/second (15 ms/frame) with a clock rate of 17ns. That works out to a rate of 1 pixel/8 cycles (the frame size was 352x288). The original algorithm was written in C, and was encapsulated in a module with a protocol where, as each pixel was input or output, a ready/valid handshake occurred.

After an initial synthesis, it was very clear that the memory architecture was the largest performance bottleneck in the design. Since the testbench could be used without modification, it was very straightforward to create multiple verified designs with multiple verified memory architectures and then create verified RTL implementations from each of them. Essentially, this architectural exploration added another dimension (memory architecture) to the design space. Since design exploration was performed on each architecture, many different RTL designs were created. In fact, at one point during the design space exploration, 10 verified RTL designs were created in a single afternoon, and over 30 were generated in total.

Figure 2 shows a graph of the performance results, in frames/second, of the various designs which were tried. Note the knee of the curve at 20 frames/second. The first 5 design points were found by setting the latency constraint on CynthHL to successively lower values until the fastest implementation was reached. At this point, memory bandwidth was the bottleneck, and each succeeding point on the graph represents a higher bandwidth memory architecture. Eventually, a memory architecture was arrived at which would support the required 66 frames/second.

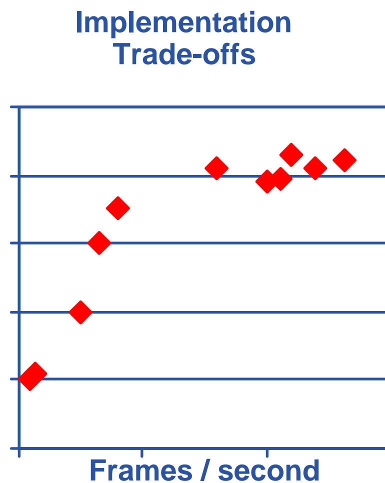

Figure 2. Performance vs. area tradeoff

Conclusion

The development of SystemC into a viable means of using a general-purpose programming language for system modeling and design is a key enabling technology for top-down, iterative system design. The development of the communication layer in SystemC along with the maturation of the register transfer layer has given C++/SystemC an effective means of expressing designs at multiple levels of abstraction. Using these classes in C++, a system model can be iteratively refined in small, continuous steps from essentially an arbitrarily high level of abstraction to an implementation.

The emerging high-level synthesis technology bridges the gap between a GPL representation and an HDL representation, and it also bridges the gap between an algorithmic level and a register-transfer level. In doing so, it enables a level of design exploration that simply wasn't possible before, improving both the time to market and the quality of the target design. This can be described as true algorithmic exploration, when not only different architectures but even different algorithms are considered for a specific design.

It is important to note that algorithmic and architectural choices are made with or without a top-down design methodology. The difference is that without integrated high-level synthesis and verification, only one algorithm, one architecture, and one implementation is ever fully considered, because there is only time to create one RTL design by hand.

With the maturation of SystemC and high-level synthesis, this will change. Designers are now able to leverage their high-level models to help evaluate trade-offs in architectures and algorithms in a way that was simply not practical before.

References

[1] "SystemC 2.0 Users Guide", Open SystemC Initiative, 2001. http://www.systemc.org

[2] "CynthHL Users Guide", Forte Design Systems, 2002. http://www.forteds.com