Hardware Synthesis from C/C++

Abhijit Ghosh, Joachim Kunkel, Stan Liao
{ghosh, kunkel, stanliao}@synopsys.com
Synopsys Inc. 700 E. Middlefield Road, Mountain View, CA, 94043 U.S.A

Abstract

Before attempting to synthesize hardware from a programming language like C or C++, we need to introduce additional semantics to be able to describe hardware behavior accurately. In particular, concurrency, reactivity, communication mechanisms, and event handling semantics need to be added. Also, a synthesizable subset of the language needs to be defined, together with synthesis semantics for programming language constructs. With these enhancements, it is possible to create C/C++ descriptions of hardware at the well-understood RTL and behavioral levels of abstraction, providing an opportunity to leverage existing, mature hardware-synthesis technology that has been developed in the context of HDL based synthesis to create a C/C++ synthesis system. In this paper, we will present some of the key ingredients of a C/C++ synthesis system and elaborate on the challenges of hardware synthesis from C/C++.

Introduction

The high level of integration provided by today's processing technology has brought new challenges in the design of digital systems, where entire systems consisting of hardware and software are being integrated into single *systems-on-chip*. This trend challenges EDA tool developers to provide tools that support the development of such systems and provide the productivity improvements required to design such systems in a cost-effective manner.

For most design flows in use today, designers at the system level start the design process by creating C/C++ models of their system. Often, the aim is to validate their algorithms and the system functionality. C/C++ is the language of choice for several reasons. Firstly, significant part of the system is implemented in software, which is often written in C/C++ and can be plugged into the model. Secondly, C/C++ provides capabilities that are beyond traditional HDL, allowing compact descriptions at a high level of abstraction. However, there is no uniform modeling style for modeling systems in C/C++ and most designers develop their own modeling style. Therefore, their models are of little use to anyone but themselves or their immediate group. In most cases, these models are thrown away and a written specification (derived from this model) is handed to the implementers of the system. A written specification is often ambiguous, open to interpretation, incomplete and inconsistent. More importantly, there is no way to verify the correctness or consistency of such a specification. Since the implementation is based on this specification, the time spent in tracking bugs and inconsistencies in the specification has a negative impact on designer productivity.

Many designers have realized the problems with the written specification and have embarked on creating ***executable specifications***. An executable specification is a model (typically in C/C++) at a high level of abstraction that captures the complete system functionality and timing (to some extent). This model is written in a well-established style so that it can be developed by one person and used by another. An accurate executable specification can provide tremendous benefits during the design of complex systems. These benefits are:

- Avoids inconsistency and errors and helps in ensuring completeness of the specification.
- Ensures unambiguous interpretation of the specification.
- Helps in validating system functionality before implementation begins.
- Helps in the creation of testbenches that can be used throughout the design process.

However, an executable specification in C/C++ solves only half of the problem of modern system design. In today's methodology, this executable specification has to be converted into an implementable HDL (Verilog or VHDL) model. There are several disadvantages to this approach. Firstly, recoding a C/C++ specification into a HDL is manual and time consuming. This process is also error prone, requiring significant effort in verifying the HDL code. Also, in today's methodology, the implementable model is at the register-transfer level, which is at a lower level of abstraction than the executable specification. Therefore, the designer has to manually create the datapath and the finite-state machine. These are time consuming tasks.

In a C/C++-based design flow, designer productivity can be significantly improved because we can eliminate translation into an HDL by synthesizing directly from C/C++ specifications. This not only reduces translation time, but eliminates bugs during translation which can take significant time to track down. In addition, we can also ease the verification bottleneck by reusing the testbenches that were developed during system validation. Finally, significant productivity gains can be obtained by synthesizing from a higher level of abstraction than register-transfer level (RTL).

The promise of increased productivity in a C/C++-based design flow has led us to develop a C/C++-based design environment, called *Scenic*. In this paper, we will present parts of this environment (discussion of the entire environment is beyond the scope of this paper). At first we will present the modeling constructs needed to model hardware efficiently in C/C++. We will then present some details on how a C/C++ specification can be synthesized. We will conclude with some observations on the challenges facing C/C++ synthesis tool developers.

Modeling Hardware in C/C++

C/C++ are software programming languages and have little support for describing hardware efficiently. To model hardware in C/C++, we need the following language features that are not present in the C/C++ languages.

Concurrency: Hardware is inherently parallel, while C/C++ programs are inherently sequential. Therefore, we introduce the notion of processes, which are programs that execute concurrently. The semantics is akin to that of communicating sequential processes. Instead of extending the language through a new syntactic construct, we encapsulate concurrency in an object or class definition. We can then build hardware processes by using subtyping and virtual function facilities of C++. Scenic processes have semantics similar to *always* blocks in Verilog or a *process* loops in VHDL.

Signals: In software, parallel processes communicate using primitives such as semaphores, critical regions, *etc*. Such primitives assume that the processes have easy access to each other's states, which is not true for hardware. Therefore, our hardware processes communicate with each other through signals (Scenic signals have wire semantics and therefore easily implementable in hardware). Once again, instead of adding new syntactic constructs, we use template classes for signals and use template instantiation to derive signals of particular types.

Reactivity: Hardware is inherently reactive, *i.e.* it is in continuous interaction with its environment. Reactivity is essential to describing hardware systems at all levels of abstraction. We implement reactivity through a hybrid of event-driven and process-driven approaches. More details on this can be found in [1].

Hardware data types: Arbitrary precision signed and unsigned integers, bit vectors and fixed point types are needed to model hardware efficiently. We model these data types as classes.

A complete description of all hardware primitives supported in the Scenic environment is beyond the scope of this paper. The following is an example of a 16-bit CRC generator described using the classes that are provided with the Scenic environment (some modeling constructs are highlighted in bold).

```
struct crc_ccitt : public sc_sync {
   const sc_signal<bool>& reset;
   const sc_signal<bool>& in;
   sc_signal_bool_vector& out;

   crc_ccitt(const char * NAME,
      sc_clock_edge& CLK,
      const sc_signal<bool>& RESET
      const sc_signal<bool>& IN,
      sc_signal_bool_vector& OUT)
   : sc_sync(NAME, CLK), reset(RESET),
     in(IN), out(OUT)
   {
      watching(reset.delayed() == 1);
   }

   void entry();
};

void crc_ccitt::entry()
{
   bool s;
   sc_bool_vector crc(16);

   if (reset == 1) {
      out = 0;
   }
   wait();

   while (true) {
      crc = out;
      s = crc[15] ^ in;
      out = (crc.range(14,12), crc[11] ^ s,
             crc.range(10,5), crc[4] ^ s,
             crc.range(3,0), s);
      wait();
   }
}
```

Synthesis from C/C++

An executable specification is generally not ready for synthesis. Refinement is the process by which just enough implementation details and constraints are added to an executable specification to make it an ***implementable specification***, *i.e.* a specification that is synthesizable and can achieve good quality of results. The amount of implementation detail added depends on the degree of control over synthesis that the user wants. Typically, the more control the users want, the closer to register-transfer level (datapath + finite-state machine) of detail they have to specify. If a higher level of abstraction, such as behavioral level is desired, then the user loses some control over the architecture because the synthesis tool selects it (though the user has some control over directing the tool towards the right architecture, e.g., by specifying constraints on resources and timing).

Refinement consists of three steps. The first step is called *data refinement*, whereby C/C++ built-in types are replaced by data types of the right precision determined by the user. One example of such refinement is the refinement of floating-point types to fixed-point types.

The second step is *control refinement*. The most important aspect of control refinement is specifying the input/output behavior of each block in the design, *i.e.* when inputs are sampled and when outputs are produced. During control refinement, a user may decide what level of abstraction she desires to synthesize from. For a behavioral level of abstraction, defining the I/O behavior and setting the design constraints is all that is required. To refine a design to RTL, the user has to create the finite-state machine and the datapath herself.

The third step in refinement is to ensure that the appropriate coding style has been followed and that all synthesizable models use constructs from the synthesizable subset. The synthesizable subset for C/C++ consists of the entire C language except the obvious constructs that do not have a well defined hardware semantics, namely dynamic memory allocation, pointers, arbitrary gotos, recursion, *etc*.

Since the C/C++ language can support various levels of abstraction, there are few restrictions on what can be specified for synthesis. The hardware semantics of C/C++ operators and expressions involving such operations are similar to that of most HDLs. Semantics of control flow statements like if-then-else, switch-case, while, *etc*. are also well defined. A structural datapath consisting of arithmetic operators can be specified as easily as a finite-state machine. A behavioral model (one without an architecture) can specified more easily.

The C/C++ synthesis tool leverages state-of-the-art and mature behavioral and RTL synthesis and logic optimization capability. The C/C++ language description is parsed into an intermediate representation on which various compiler optimizations like constant propagation, common sub-expression elimination, *etc*. are performed. Existing work in parallelizing compilers can be leveraged to discover loops that can be parallelized or pipelined, and memory accesses that can be disambiguated (*e.g.*, it may be possible to determine that `a[i]` and `a[j]` do not access the same memory location), thereby eliminating unnecessary dependencies and increasing the throughput of a design. Depending on the level of abstraction of the input description, the intermediate representation is either converted into a control-data-flow graph that can be used for behavioral synthesis, or is converted into a form that a hardware generator (also called netlister) can use to generate hardware for RTL synthesis. Existing RTL and behavioral synthesis algorithms can then be leveraged to produce quality of results that are comparable to those obtainable through traditional means.

Apart from being used for ASIC implementation, such a synthesis tool can provide other benefits. It can be used with low optimization effort for area/speed estimation. It can also be used for exploration of the architecture of hardware blocks.

Challenges

For the most part, synthesis from C/C++ descriptions is similar to synthesis from HDL descriptions, which is a mature technology today. However, C/C++ presents some unique challenges for synthesis. Firstly, synthesis of descriptions using pointers is difficult. Due to effects such as aliasing, analysis of pointers and where they point to is non-trivial. Though one can restrict synthesizable programs to exclude pointers, this problem cannot be ignored in the long term because users will demand the flexibility of pointers. There is some progress in this area (see [2]), though more work needs to be done.

C/C++-based synthesis will enable a new set of designers currently unfamiliar with HDLs to design hardware. Some of these designers will have more experience in software than in hardware, especially in synthesis and synthesis friendly coding styles. The challenge for the tool developer is to provide a coding style checker that will check not only for the synthesizable subset, but will check for coding styles that produce poor quality of results. This is a difficult problem because a “good” coding style is hard to quantify.

To increase designer productivity further, parts of the refinement process, in particular data refinement, should be automated. Existing work in floating-point to fixed-point refinement [3] should be extended to cover integral data types.

Another challenge is in extending the synthesizable subset to include object-oriented features like virtual functions, multiple inheritance, *etc*. More work is required to define the object-oriented semantics for hardware before synthesis can be attempted.

Finally, C/C++-based synthesis will be acceptable only if the entire flow works. This means that a complete verification flow and interoperability with existing HDL models has to be established before C/C++ synthesis tools get adopted.

References

[1] S. Liao, S. Tjiang and R. Gupta, “An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment”, *Design Automation Conference*, pp. 70-75, June 1997.

[2] L. Semeria and G. De Micheli, “SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C”, *International Conference on Computer-Aided Design*, pp. 340-346, November 1998.

[3] M. Willems, V. Bursgens, H. Keding, T. Groetker, H. Meyr, “System Level Fixed-Point Design Based on an Interpolative Approach”, *Design Automation Conference*, pp. 293-298, June 1997.