Computer Science Education
Vol. 17, No. 2, June 2007, pp. 87–96

Routledge
Taylor & Francis Group

Simulation, Design Abstraction, and SystemC

Ed Harcourt*
St Lawrence University, USA

SystemC is a system-level design and simulation language based on C++. We've been using SystemC for computer organization and design projects for the past several years. Because SystemC is embedded in C++ it contains the powerful abstraction mechanisms of C++ not found in traditional hardware description languages, such as support for object-oriented programming and generic programming (templates). This support for abstraction allows instructors to reinforce standard abstraction concepts such as information hiding, interfaces, and abstract data types, standard fare in a computer science curriculum. Furthermore, embedded software is often written in C++ and SystemC provides threading facilities useful for designing and implementing embedded software.

1. Introduction

SystemC is a C++ class library that provides a rich set of data types useful for designing, specifying, and simulating hardware/software systems at a variety of levels of abstraction, as described in Grotker, Liao, Martin, and Swan (2002). SystemC has wide support from industry and has become an important design language for system-level design. Information about SystemC, including the *Language Reference Manual*, can be found at Open SystemC Initiative (2006).

We've been using SystemC for senior capstone projects and consider it ideal for several reasons, one of which is that SystemC is easier to learn than other hardware description languages (HDLs), such as Verilog and VHDL, because it is embedded in C++, a language familiar to most computer science students. In computer science we stress abstraction, information hiding, and interfaces. Because SystemC is embedded in C++ its abstraction facilities are more powerful than the abstraction facilities available in existing HDLs. These abstraction facilities allow us to reinforce a basic principle of hardware design (and software design), that a circuit has a low level implementation and a higher level black-box behavior and that each of these levels

*Department of Mathematics, Computer Science, and Statistics, St Lawrence University, Canton, NY 13617, USA. E-mail: edharcourt@stlawu.edu

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/07/020087-10
© 2007 Taylor & Francis
DOI: 10.1080/08993400601165248

can be implemented in SystemC and support an identical interface. See Harcourt (2005) for a discussion of how SystemC compares with other simulation technologies.

SystemC also comes with a rich set of predefined hardware data types. SystemC is flexible; student projects have ranged from building low level hardware components at the gate level all the way to high level instruction set simulators. Additionally, SystemC's support for concurrency makes it a good choice for embedded software projects.

1.1. Language-Based Design

SystemC is a hardware description language, although SystemC goes beyond hardware and can also be used for developing embedded system software. Consequently, SystemC is often called a system-level design language useful for designing, implementing, and simulating both the hardware and the software portions of a digital system. Language-based design refers to the use of textual languages that contain dedicated constructs for designing and simulating hardware. Contrast this with graphical schematic capture tools where circuits are drawn on a screen similar to the way they would be drawn on paper.

Language-based environments are usually based on the hardware description languages (HDLs) Verilog and VHDL, with SystemC being a relatively new entry. Verilog, VHDL, and SystemC are IEEE standards and are used heavily in industry. Many tools are available for designing, testing, debugging, and synthesizing an HDL-based design. Language-based design is even used in the popular textbook by Patterson and Hennessy (2005).

Analogous to a chip, a block of hardware is visualized as a black box with a pin interface. HDLs allow the designer to capture this block-oriented nature of hardware, where systems are composed of interconnected blocks communicating through wires and wires are connected to pins on blocks. In HDLs a block is called a module, a pin is called a port, and a wire is called a signal.

The main strengths of HDLs is their support for large designs (scalability) and abstraction. HDLs provide many of the same programming constructs found in standard imperative languages, such as loops, conditionals, and functions. These constructs are generally used in limited ways depending on the level of abstraction of the design. For example, in a gate-level design a student would not normally use any looping or conditional constructs, whereas a higher level design might make judicious use of the full language.

Support for abstraction in traditional HDLs lags behind the kinds of abstraction mechanisms found in modern programming languages. For example, support for classes, polymorphism and generic programming is rudimentary or non-existent in Verilog and VHDL. Consequently, it is common for higher level abstract designs to be developed in C++ first. SystemC brings together the hardware concepts found in standard HDLs (modules, ports, wires, bits) along with the powerful abstraction mechanisms found in C++.

2. SystemC

We briefly introduce SystemC using a canonical example of a full adder. In the next section we provide further examples of abstraction, followed by a discussion of how SystemC is used to support teaching computer organization and design. We only introduce enough SystemC to discuss the examples. The reader should consult the SystemC language reference manual for a complete description of SystemC. The language reference manual can be downloaded from www.systemc.org.

SystemC allows the user to express block structure through the use of predefined classes that represent blocks (modules), wires (signals), and pins (ports). SystemC also contains a large library of useful hardware data types, such as bits, bit vectors, four valued logic types, fixed point types, and sized integers. Furthermore, since SystemC is a C++ library, SystemC development tools are just the C++ compilers, debuggers, and development environments that students are already familiar with.

2.1. A Simple Example

The best way for a student to learn SystemC is to experiment with simple combinational logic. For example, the following logic equations describe the Sum and Carry bits of a full adder. These equations have an immediate and obvious implementation in terms of gates.[1]

$$\mathrm{Sum} = a \oplus b \oplus c$$
$$\mathrm{Carry} = ab + ac + bc$$

Constructing the adder in SystemC in terms of gates we first create a module named Adder by declaring a class that derives from the SystemC class sc module (Figure 1, line 1). Modules can have input ports, output ports, and input/output ports declared

```
1 class Adder : public sc_module {
2 public:
3    sc_in<bool> a, b, c; // inputs
4    sc_out<bool> sum, carry; // outputs
5    sc_signal<bool> tmp1, tmp2, tmp3; // internal wires
6
7    SC_CTOR(Adder) {   // Wire gates together
8       x1(a,b, c,sum); //sum=axorbxorc
9       ab(a, b, tmp1); // tmp1 = a && b
10      ac(a, c, tmp2); // tmp2 = a && c
11      bc(b, c, tmp3); // tmp3 = b && c
12      o1(tmp1, tmp2, tmp3, carry); // carry = tmp1 || tmp2 || tmp3
13 }
14
15 private:
16    Xor x1; // 3-input xor-gate
17    And ab, ac, bc; // three and-gates
18    Or3 o1; // 3-input or-gate
19 };
```

Figure 1. An adder constructed from And, Or, and Xor gates

using the classes sc_in, sc_out, and sc_inout. These port classes are parameterized with the type of value communicated on the port. In this example ports are single bits which we represent using the C++ type bool (lines 3–4). Line 5 declares internal signals needed to wire together the gates. The constructor (lines 7–13) ties together all of the gates using the signals, connecting the outputs of the OR gate and the XOR gate to the output ports of the adder. Lines 16–18 declare the instances of the gates needed to implement the circuit.

2.2. Simulating the Design

Every simulation technology needs a way to provide inputs to the design and observe outputs. In language-based designs we test designs by constructing a test bench, another module that instantiates the design under test. Our example test bench for the adder provides inputs (stimulus) at regular intervals that are controlled by a clock. Figure 2 shows a simple test bench for the adder and a main program that instantiates the adder and runs the simulation for 10 time units. The test bench generates random stimulus to the adder on every clock pulse. The clock pulse is provided by a predefined SystemC sc_clock module.

The test bench also introduces SystemC's event-based style of programming, where a module listens for events on ports. Lines 9–10 in Figure 2 declare that the event handler named run should execute whenever there is a rising edge on the clock signal.

3. Teaching Abstraction

Hardware examples provide the perfect material to reinforce the importance of abstraction and interfaces, concepts stressed heavily in a computer science curriculum.

```
 1 class TestBench : public sc_module {
 2 public:
 3    sc_clock clock;
 4    Adder fa;
 5    sc_signal<bool> x, y, z, sum, carry;
 6
 7    SC_CTOR(TestBench) {
 8       fa(x, y, sum, carry);
 9       SC_METHOD(run);
10       sensitive_pos << clock;
11    }
12
13    void run() {
14       x.write(random_bool());
15       y.write(random_bool());
16       z.write(random_bool());
17    }
18 };
19
20 // run the simulation for 10 clock cycles
21 int sc_main(int argc, char* argv[]) {
22    TestBench tb("tb");
23    sc_start(10);
24 }
```

Figure 2. A module that tests the adder

A circuit is a black box with an interface; how that interface is implemented may or may not be important. For example, a useful exercise is for students to implement simple combinational circuits in terms of gates and then re-implement those circuits at a higher level where the functionality of the circuit is described in terms of programming language constructs. This also provides an opportunity to talk about higher level descriptions as executable specifications of lower level designs.

Hardware engineers classify designs at various levels of abstraction. At the lowest level, the gate level, hardware systems are designed in terms of digital logic. At the next level, the register transfer level (RTL), hardware systems are designed in terms of interacting memory elements and functional units (registers, ALUs, pipeline stages, etc.). At the highest level of abstraction, the behavioral level, hardware is specified in terms of sequential algorithms.

As an example, rather than implement an adder in terms of gates we can construct the adder at a higher level in terms of SystemC data types and C++ language constructs. Without showing the entire module the following code snippet declares a two bit value using the SystemC data type sc_uint, calculates the sum of the inputs, and then writes the outputs.

```
sc_uint<2> result=a.read()+b.read()+c.read();
sum.write(result[0]);
carry.write(result[1]);
```

The point to stress to students is that this implementation of an adder has the identical interface as the gate implementation and that the two versions of the adders are "plug compatible."

Another canonical example that highlights the role of abstraction is a sequential circuit that implements a D flip-flop (a one bit memory element). An obvious gate-level implementation uses two cross-coupled NOR gates with some additional logic to control the clock. More illuminating and descriptive for the student is a module that describes the external behavior of the flip-flop, which simply specifies that the output takes on the value of the input on the rising edge of the clock, as shown in the module DFF below.

```
class DFF : public sc_module{
public:
  sc_in<bool> d, clk;
  sc_out<bool> q;
  SC_CTOR(DFF){
  SC_METHOD(run);
  sensitive_pos <<clk;
  }

  void run(){q.write(d.read());} // q=d
  };
```

3.1. Programming Abstraction

Another kind of abstraction familiar to computer scientists but not available in traditional HDLs uses the advanced features of the programming language, in this case C++, to write generic, reusable components. For example an n bit ripple carry adder can be constructed from n full adders. One way to do this is to use an array of adders, an array of ports, and templates. Figure 3 shows the complete ripple carry adder. It takes a non-type template parameter that specifies the size of the adder. Alternatively, the higher level specification of this adder would just use addition on n bit quantities rather than instantiating individual adders.

Leveraging C++ abstraction features, such as function operator overloading, templates, and class hierarchies, a user can take abstraction to the extreme. For example, the general concept of a gate is that it has n inputs and computes some function as an output. It is straightforward to write a generic gate module paramaterized on the number of inputs and the function it computes, all passed in as template parameters (Figure 4). This example also uses the built-in for_each algorithm from the Standard C++ Library. The for_each algorithm applies a function to every item in a list, accumulating and returning the final result.

3.2. System Level Models

When an engineer is initially designing a system they may not know which subsystems will be in the form of hardware and which will be in the form of software. Early in the design cycle designing in terms of bits, gates, or adders may be premature. At the

```
template <int n>
class Adder : public sc_module {
public:
   sc_in<bool> a[n], b[n]; // two n-bit inputs
   sc_out<bool> s[n];      // the n-bit output
   sc_out<bool> Cout;      // the carry out
   sc_signal<bool> c[n];   // internal carry signals
   FullAdder *adder[n];    // array of n adders

   SC_CTOR(Adder) {

      // instantiate n full adders.
      for (int i = 0; i < n; ++i)
         adder[i] = new FullAdder();

      // Set up bits 1 through n -2
      for (int i = 1; i < n -1; ++i) {
         adder[i]->a(a[i]);     // Tie input a[i] to the
         adder[i]->b(b[i]);     // a-input of adder i.
         adder[i]->c(c[i-1]);   // The same for input b[i].
         adder[i]->sum(s[i]);   // Tie the carry-out of adder i
         adder[i]->carry(c[i]); // to the carry-in of adder i+1
      }
   }
};
```

Figure 3. An n-bit ripple carry adder

```
 1 template<typename F, int n>
 2 class gate : public sc_module {
 3 public:
 4    sc_in<bool> in[n]; // n inputs
 5    sc_out<bool> out; // one output
 6
 7    SC_CTOR(gate) {
 8       SC_METHOD(run);
 9
10       for (int i = 0; i < n; ++i) // make the gate sensitive
11          sensitive << in[i]; // to all of the inputs
12       }
13
14    void run() {
15       F v = for_each(in, in + n, F());
16       out.write(v.value());
17    }
18 };
```

Figure 4. A generic gate module

system level we design in terms of transactions, as opposed to bit-level changes on ports. To do this we use the property that ports in SystemC can carry a value of any type T. Returning to the D flip-flop example, rather than have ports carry a single bit they can be declared to carry values of type sc int<32>, thereby creating a 32 bit register, or more abstractly a double if the designer cannot yet commit to a floating point representation.

Ports and signal types need not be limited to simple scalar data types. Consider the following type that defines a network packet.

```
struct NetworkPacket{
  char header[10];
  char body[255];
};
```

A port declaration can then use NetworkPacket as the type of a port; sc_in <NetworkPacket>. In this manner system level models are neutral as to whether they will be targeted at hardware or software.

Another kind of abstraction addresses the way in which modules communicate. In SystemC the default communication model is a discrete event where modules communicate and synchronize through events generated by writing to ports. This model matches closely the way in which hardware communicates. SystemC allows the user to change the semantics of communication with their own communication model. For example, a high level model that instantiated a hardware FIFO may be better suited to use blocking reads and writes to the FIFO rather than a lower level protocol defined in terms of pins and events. Abstracting the communication model in this way is unique to SystemC and is not supported at all in traditional HDLs. SystemC further supports system-level design by providing software threads, also not found in traditional HDLs.

3.3. Debugging

Debugging event-driven programs is more complicated than debugging sequential programs. Nevertheless, since SystemC programs run in standard C++ development environments a C++ debugger can be used to step through designs, set breakpoints in modules, and view the values on signals and ports.

The values of ports and signals vary over time. Tools for viewing these value changes are called waveform browsers. SystemC provides a trace capability that outputs value changes on ports and signals to a value change dump (VCD) file, a standard IEEE file format supported by all waveform browsers.

4. Student Projects

In this section we describe how we have used or plan to use SystemC to support student hardware/software projects.

4.1. Combinational and Sequential Circuits

An obvious place to start is with the plethora of examples that come from combinational circuits. Useful projects include having students implement basic components such as multiplexors, decoders, and ripple carry adders and then using these building blocks to construct ALUs.

Sequential circuit examples include canonical control-oriented circuits, such as vending machines and traffic light controllers. For both combinational and sequential designs the student should consider designing both a high level specification of the circuit and then a lower level implementation. For sequential circuits a high level design might use a switch statement and an enumeration to control state changes of the finite state machine, with the low level implementation using flip-flops and combinational logic. Implementing these two levels of a sequential machine highlights the often confusing concept that, at a lower level, the contents of the flip-flops hold the value of the current state.

4.2. Processor Models

The textbook by Patterson and Hennessy (2005) describes in detail three increasingly complex versions of a MIPS microprocessor. The first version is a mostly combinational single cycle version. The second version is a more complicated multi-cycle version where instructions execute in differing numbers of cycles. The third version uses an instruction pipeline. Senior projects have involved exploring each of these designs at various levels of abstraction.

Implementing the MIPS five stage instruction pipeline in SystemC is challenging, but a reasonable semester-long project. In order to complete the project it forces the

student to think and implement components at a higher level (e.g. ALUs and registers) rather than in terms of gates, as well as using good programming practice in writing reusable components.

Another nice feature of SystemC is that it is straightforward to package other software tools with the design. For example, in one project a student implemented a simple assembler, allowing the user of the processor model to enter assembly code rather than machine code. One student even implemented a web interface for the MIPS simulator.

4.3. Other Projects

Other projects not yet done but well suited for SystemC include generic cache models supporting various cache configurations, such as set associative, direct mapped, and fully associative, as well as various block replacement strategies (e.g. random, LRU). Using address traces, students explore the efficacy of different configurations.

SystemC's concurrency primitives make it a good choice for students to explore concurrency issues and implement classic concurrent programming problems, such as semaphores, producer/consumer, and even network protocols.

SystemC lends itself well to addressing programming in the large. Real hardware designs are large and complex, but subsystems generally have well-defined interfaces. Having student teams break down a large design into subsystems and then design, implement, document, and tie these subsystems together highlights the importance of interfaces and programming in the large.

5. Conclusions

SystemC is the first design language to attract wide support that enables both the hardware and software portions of a digital system to be designed using a single, common standard language. SystemC is well suited to student projects requiring mature programming skills, provides a programming environment students are already familiar with, and support for various levels and kinds of abstraction, generic programming, and object-oriented design. All of the projects undertaken to date have centered around processor modeling, but SystemC is well suited to a variety of hardware and embedded software related projects.

Note

1. The motivation for designing SystemC was not for describing gate-level circuits. Most would consider it an abuse to use it in this manner. However, smaller gate-level circuits are canonical and provide simple examples that highlight the capabilities of SystemC, as well as being easy to understand.

References

Grotker, T., Liao, S., Martin, G., & Swan, S. (2002). *System design with SystemC.* Boston: Kluwer.

Harcourt, E. (2005). Teaching computer organization and architecture using SystemC. *The Journal of Computing Science in Colleges*, *21*(2), 27 – 39.

Open SystemC Initiative. (2006). *SystemC.* Retrieved March 30, 2007, from www.systemc.org

Patterson, D., & Hennessy, J. (2005). *Computer organization and design, The hardware/software interface.* San Francisco, CA: Morgan Kaufmann.