Energy Estimation with SystemC: A Programmer's Perspective

S. XANTHOS, A. CHATZIGEORGIOU, G. STEPHANIDES
Department of Applied Informatics
University of Macedonia
156 Egnatia Str., 54006 Thessaloniki
GREECE
spiros@java.uom.gr, achat@uom.gr, steph@uom.gr

Abstract: **-** A modification to the SystemC library to enable power estimation of digital systems built upon a set of primitive logic gates is proposed. Acknowledging both the intense requirement for low power systems as well as the increasing use of SystemC as a modeling methodology, an approach for obtaining the dynamic power consumption of SystemC modules is presented. In order to correctly handle glitches during energy estimation, a simulation approach based on guarded evaluation is used. Emphasis is given to the fact that extensions to SystemC can be performed in a simple manner broadening the design and analysis possibilities of circuit designers. Even computer science students, with limited background on digital electronics, can easily grasp the concept of energy consumption and implement enhancements to SystemC, justifying its use as a common modeling platform between HW and SW designers.

Key-Words: - SystemC, energy consumption, low power, simulation, C++, modeling, object-oriented languages

1. Introduction

SystemC is becoming widely accepted as a platform for modeling systems consisting of both hardware and software components, as in the case of system-on-chip (SoC). The increasing trend towards the use of C/C++ language as a unified modeling tool and for building *executable specifications* drives the momentum behind the adoption of SystemC against other hardware description languages (HDLs) such as VHDL or Verilog. Equally important to the ability to model both HW and SW components [1], is the fact that most college graduates in the disciplines of engineering and computer science, are already familiar with object-oriented programming languages such as C++ [2]. In contrast, familiarization with other HDLs requires a significant investment in time and effort.

However, SystemC, which is essentially a library of C++ classes, lacks so far a modeling environment in which a number of simulation or synthesis possibilities exist [3]. An obvious requirement for a modeling methodology in the embedded systems domain would be to provide the power consumption of the system that is being developed, at least when low level implementation details have been addressed.

SystemC, as an object-oriented approach to modeling systems, is easily modifiable in order to augment its capabilities. Due to the hierarchical nature of the classes that support the specification of a digital circuit, it is relatively simple to insert appropriate methods for calculating the dynamic power dissipation at each node of the circuit and thus the power consumption of the whole digital system. However, one of the issues that has to be resolved when simulating a system with SystemC, is the appropriate handling of spurious transitions, which appear on the circuit nodes, due to the finite propagation delay of logic blocks. An approach for evaluating energy at specific time points and simulating the circuit employing a three-clock scheme is proposed in this paper.

The main philosophy behind object-orientation is to map, with a one-to-one correspondence, the object structure of a problem domain into an object structure of a software model. The object structure in the software model can be analyzed and annotated with information and in this way object models are extendible without disrupting the existing tools [4]. A similar approach to the one proposed in this paper for SystemC, can also be followed for other object-oriented environments for hardware specification, such as Embedded C++ [5]. The analysis and design of large scale object-oriented HW/SW systems can also be supported by modeling methodologies such as UML [6], [7].

Obviously, a programming language like C++ lacks the semantics to capture energy-related information. However, object-oriented library-based platforms can be easily augmented: modifications to SystemC have been performed by computer science students given only the necessary information concerning the power consumption of digital circuits. Limited background on electrical engineering concepts was not a hindering factor for building models upon SystemC and extending its features.

The rest of the paper is organized as follows: In section 2 the way in which power consumption is calculated is presented along with some assumptions that are made for clarity reasons. Section 3 describes the necessary modifications in the declaration of modules while in section 4 the modifications to the SystemC library classes are discussed. Section 5 presents our approach in handling glitches in SystemC. Finally, we conclude in section 6.

2. Power of SystemC modules.

Dynamic power consumption in digital circuits, accounts for the largest portion of the total power and is due to the energy that is drawn from the power supply to charge parasitic capacitors [8]. In this study, it is assumed that these parasitic capacitances are made up mainly of gate and diffusion capacitances. For a node in a digital circuit with capacitance C_L that undergoes a transitions from logic "0" to logic "1" during a period T, the energy that is drawn from the power supply during the same period is given by:

$$E_{dyn} = aC_L V_{DD}^2 \qquad (1)$$

where V_{DD} is the power supply voltage.

Assuming that each node in a digital circuit will be both an input and an output of a logic gate (except for primary input or circuit output nodes), the capacitance being switched consists of two components: First, the output capacitance C_{out} of the previous level logic gate which consists of the diffusion capacitances of all MOS transistors connected to the output node of the driving gate (Fig. 1). This capacitance depends on the transistor sizes and the technology used and for the rest of this paper will be assumed constant for all logic gates of the same type. Second, the input capacitance C_{in} of the next level logic gate (Fig. 1), which consists of the gate capacitances of the input signal receiving transistors. These capacitances depend on the operating region of the transistors, but for the sake of simplicity their value will be assumed constant and equal to an average value for the rest of the paper ($C_g = C_{ox}WL + C_{overlap}$) [9]. Since each input in a CMOS digital circuit is connected to a pair of nMOS and pMOS transistors, a constant input capacitance will be assigned to each input (assuming that all transistors of the same type in the circuit have the same size which is reasonable for datapath circuits). A final assumption being made, is that each logic gate has a single output.

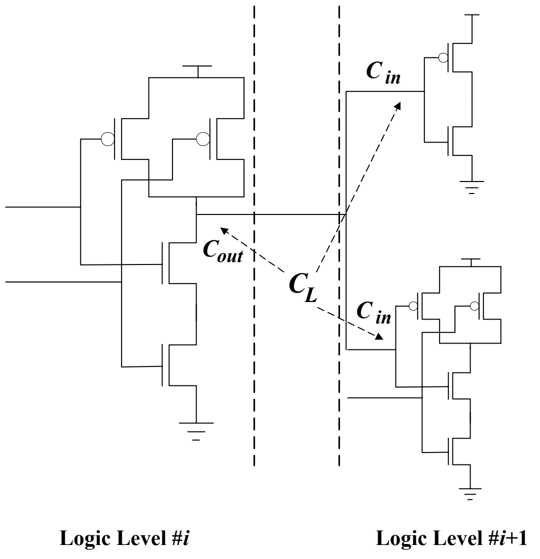

Fig. 1: Node capacitance components

For a simulator built upon the SystemC library it is then sufficient to count for each module implementing a logic gate the number of zero-to-one transitions at its input and output nodes, calculate the corresponding energy and register this energy consumption to a global variable. In the next section, it will be demonstrated that such an extension to SystemC classes is relatively simple, justifying the importance of a common modeling platform between HW and SW designers, based upon a standard programming language such as C++.

3. Module declaration

All building blocks in SystemC are objects of a container class, which is part of the SystemC library and is called a *module*. This hierarchical entity can have other modules or processes contained in it [10]. Having a module whose operation is specified by means of a process corresponds to a behavioral description, while specifying its functionality by means of other sub-modules corresponds to a structural description. As an example of behavioral description consider the following NAND2 gate:

```
// Filename : nand2.h

#include "systemc.h"

#ifndef NAND2_H
#define NAND2_H

SC_MODULE(nand2)
{
       sc_in<bool> A;
       sc_in<bool> B;
       sc_out<bool> F;

       sc_in<bool> eval;
       int inX;
       int outX;
       bool PrA,PrB,PrF;
       void do_nand2();
       void evaluate();

       SC_CTOR(nand2)
       {
              inX=0;
              outX=0;
              PrA=false;
              PrB=false;
              PrF=false;
              SC_METHOD(do_nand2);
              sensitive << A << B;
              SC_METHOD(evaluate);
              sensitive_pos << eval;
       }
};

#endif
```

With boldface are indicated those parts of the NAND2 definition that are required for enabling energy estimation at the gate level. The purpose of the input port *eval* that has been added, is to enable energy estimation on positive edges of the signal that is connected to *eval* port. Integer variables *inX* and *outX* hold the number of zero-to-one transitions at the inputs and output of the gate, respectively. *SC_METHOD* is one of the process types that SystemC supports and is scheduled whenever an event occurs on a signal that the method is sensitive to [10]. Processes are used to model concurrency and are the basic units of execution in SystemC. Process *SC_METHOD*(do_nand2), which is made sensitive to inputs *A* and *B*, is not altered and is the one in which the output of the gate is evaluated. Process *SC_METHOD*(evaluate), which is responsible for calculating the energy consumption of the gate, is sensitive to positive edges of the signal that is connected to *eval.* Consequently, method *evaluate*( ) will be called each time the *eval* input value changes from zero to one [11]. It should be noted that signals *A*, *B* and *F* are objects of template classes *sc_in* and *sc_out* rather than simple boolean variables, and as such their values should be accessed by appropriate public methods of the corresponding classes.

The implementation file of the NAND2 gate is shown below:

```
// Filename : nand2.cpp

#include "nand2.h"

void nand2::do_nand2()
{
   F.write( ! (A.read() & B.read()) );
}

void nand2::evaluate()
{
   outX=0;
   inX=0;
   bool Ain, Bin, Fout;
   Ain = A.read();
   Bin = B.read();

   Fout = !(Ain & Bin);

       //Calc. zero-to-one transitions
   if( Ain == true && PrA == false )
     inX++;
   if( Bin == true && PrB == false )
      inX++;
   if( Fout == true && PrF == false )
      outX++;

   //Register energy dissipation
   setEnergy(inX, outX, "NAND2");
   //Store current values
   PrA = Ain;
   PrB = Bin;
   PrF = Fout;
}
```

The implementation of the *evaluate*( ) method, calculates the number of zero-to-one transitions at its inputs and output. Boolean variables *Ain*, *Bin* are used in conjunction with method *read*( ) for accessing the values of the input signals, while the value of the variable *Fout* is evaluated. The use of *read*( ) and *write*( ) methods for accessing signal values is also consistent with the concept of delta cycles [11], according to which signals are not updated until all threads have executed. To register the dissipated energy, a call to method *setEnergy* is made, with arguments the number of input and output transitions, as well as a string indicating the type of the gate. Method *setEnergy* is defined in the *sc_module* class of the SystemC library and will be described later. Finally, the input and output values are stored to serve as previous input and output values during the next evaluation of the gate output.

As an example of a module declared in a structural fashion consider the following EXOR2 gate consisting of 4 NAND2 gates:

```
// Filename : exor2.h

#include "systemc.h"
#include "nand2.h"
#ifndef EXOR2_H
#define EXOR2_H

SC_MODULE(exor2)
{
   sc_in<bool> A;
   sc_in<bool> B;
   sc_in<bool> eval;
   sc_out<bool> F;

  //4 nand2 instances within exor2
  nand2 n1;
  nand2 n2;
  nand2 n3;
  nand2 n4;
  //internal signals within exor2
  sc_signal<bool> S1;
  sc_signal<bool> S2;
  sc_signal<bool> S3;

  SC_CTOR(exor2): n1("N1"), n2("N2"),
                  n3("N3"), n4("N4")
  {
       //structural description
       n1.A(A);
       n1.B(B);
       n1.F(S1);
       n1.eval(eval);

       n2.A(A);
       n2.B(S1);
       n2.F(S2);
       n2.eval(eval);

       n3.A(S1);
       n3.B(B);
       n3.F(S3);
       n3.eval(eval);

       n4.A(S2);
       n4.B(S3);
       n4.F(F);
       n4.eval(eval);
  }
};

#endif
```

In the definition of a module employing a structural description, the only changes concerning energy consumption are the addition of the *eval* port, since energy dissipation will be calculated at the modules specified at lower hierarchical levels. As it becomes obvious, the distribution of the *eval* signal, resembles a global clock network, which however is used only for simulation purposes and helps in handling correctly glitches, as it will be shown in section 5. The EXOR2 implementation file contains nothing since the definition of the gate is structural.

4. Modification to the SystemC library

To enable each gate defined in the library to register its energy dissipation, the following trivial modification to the *sc_module* class of the SystemC library is required (addition of one method):

```
// Filename: sc_module.h

#ifndef SC_MODULE_H
#define SC_MODULE_H
#include "systemc/utils/sc_EnergyHandler.h"
 . . .
class sc_module: public sc_object {
 . . .
public:
   void setEnergy(int inX, int outX,
                  char *);
   . . .
};

#endif
```

The implementation of this method is shown below:

```
void sc_module::setEnergy(
                      int inX,
                      int outX,
                      char *gateType)
{
   sc_EnergyHandler::getEnergyHandler()
     -> setEnergy(inX,outX, gateType);
}
```

Finally, since each module implementing a primitive logic gate has to register the dissipated energy on each positive edge of signal *eval*, an object has to be added in order to hold the total energy dissipation of the system centrally. For this reason, a separate class called *EnergyHandler* has been added to the SystemC library. This class, is a singleton class, which is a creational pattern in C++ assuring a maximum of one objects of its type at any given time and a global access point to the created object [12]. Assuming for simplicity that the number of available gate types is three (INV, NAND2, OR2) the header file for the *EnergyHandler* class is:

```
// File name: sc_EnergyHandler.h

class sc_EnergyHandler {
public:
   static sc_EnergyHandler*
                get_EnergyHandler()
   {
     static sc_EnergyHandler m_EnergyHndlr;
     return &m_EnergyHndlr;
   }

protected:
   sc_EnergyHandler() { };

public:
   void setEnergy( int inputX, int outX,
                char* gateType);

   double getEnergy();

private:
   double energy;
   float Cout[3], Cin;
};
```

The constructor of the class is declared as protected member to avoid direct object creation. The static method *getEnergyHandler*( ) returns a pointer to the static member *m_EnergyHndlr*, which in turn is an object of the class *EnergyHandler*, instantiated only during the first call of *getEnergyHandler*( ).

Method *getEnergy*( ) returns the value of private member *energy* to the top-level routine *sc_main*( ) or to an appropriate *monitor* module.

In the implementation of *EnergyHandler*, the constructor initializes the input and output capacitances associated with each gate and also initializes the energy consumption to zero. If required, appropriate *set* methods can be provided, for user settings of additional parameters such as capacitance values or power supply voltage.

```
// File name: sc_EnergyHandler.cpp

#include "sc_EnergyHandler.h"

#define INV 0
#define NAND 1
#define OR 2
#define VDD 2.5

sc_EnergyHandler::sc_EnergyHandler( )
{
  energy = 0;
   //Output and Input Capacitances (fF)
  Cout[INV] = 6;
  Cout[NAND] = 7.5;
  Cout[OR] = 11.02;
  Cin = 20;
}

void sc_EnergyHandler::setEnergy
   (int inX, int outX, char* gateType)
{
switch (gateType)
 {
   case "INV":

       energy = energy+inX*Cin*VDD*VDD
              +outX*Cout[INV]*VDD*VDD;
        break;
   case "NAND":

        energy = energy+inX*Cin*VDD*VDD
             +outX*Cout[NAND]*VDD*VDD;
        break;
   case "OR":

       energy  =energy+inX*Cin*VDD*VDD
             +outX*Cout[OR]*VDD*VDD;
        break;
   default:
        energy = -100000;
        break;
   }
}

double sc_EnergyHandler::getEnergy()
{
   return energy;
}
```

The implementation of method *setEnergy* calculates the energy consumption according to eq. (1) and using the corresponding capacitance values for each gate type.

5. Handling of Glitches

The inherent property of processes in SystemC to execute once activated until they return control to the simulation kernel [1], [11], inserts a finite propagation delay from one logic block to the next. This nonzero propagation delay, which is a characteristic of actual circuits as well, causes spurious transitions or *glitches* to occur [9]. In other words, a node can exhibit multiple transitions before settling to the correct logic level. From an energy consumption point of view, the glitches contribute to the total power dissipation. However, they are only partial transitions and therefore their contribution is limited compared to normal rail-to-rail transitions.

A simplistic approach in counting zero-to-one transitions using a SystemC model would erroneously take also into account these glitches as full rail-to-rail transitions. For example, if energy estimation is performed within the main method of each gate, which is sensitive to the gate inputs, would lead to erroneous results. If a model is to be used for energy calculation, then it should take into account that SystemC does not provide a mechanism for ignoring glitches. The approach that we have followed in order to consider only full-rail zero-to-one transitions, is based on estimating the energy consumption only after the nodes have settled to their final logic level for each input vector. This post-calculation energy estimation is enabled by adding to each module the separate signal called *eval.*

In this way and by using a three-clock scheme for simulating the circuit, each module calculates its energy dissipation only during positive edges of signal *eval* and this is performed after the application of an input vector. A typical simulation testbench for SystemC consists of a *stimulus generator* (*stimgen*), the *circuit under test* (CUT) and a *monitor* module for displaying results (Figure 2).

To assure that evaluation of the circuit output is performed at an earlier time than energy estimation, three clock signals are generated as shown in Fig. 3. During the positive edge of *clock*1 the *stimgen* module presents the input vector at its outputs and the *CUT* evaluates. Energy evaluation is performed on the positive edge of *clock*2, which causes all *evaluate* processes to resume operation. Finally, the results are displayed on the positive edge of *clock*3. The purpose of the signals is only to define non-simultaneous time points for each operation.

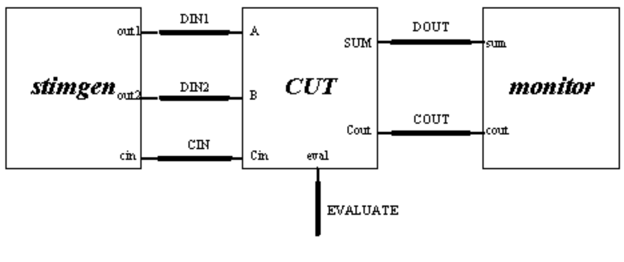

Fig. 2: Simulation testbench with *Evaluate* signal

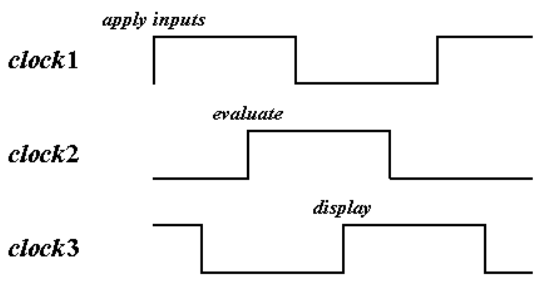

Fig. 3: Synchronizing clock signals

Implementing the full adder circuit shown in Fig. 4 employing the above modules in SystemC and modifying the SystemC library as described, gives the results shown in Fig. 5 after running the simulation. AND gates will be implemented as a NAND2 gate in series with an inverter and therefore will include one additional internal node. The same holds for the EXOR2 gates, which will include 4 internal nodes. The *Energy* column shows the energy that has been dissipated during each cycle.

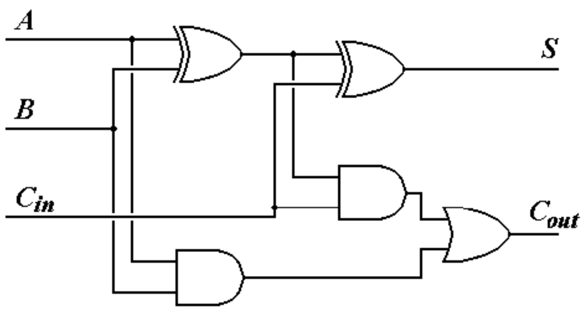

Fig. 4: Full Adder

SystemC 2.0.1 --- Aug 30 2002 10:41:15
Copyright (c) 1996-2002 by all Contributors
ALL RIGHTS RESERVED

InA	InB	Cin	S	Cout	Energy(fJ)
0	0	1	1	0	*depends on init state*
1	0	0	1	0	968.75
0	1	0	1	0	546.88
1	1	1	1	1	1325.00
0	0	1	1	0	468.79

Table 1: Full Adder simulation results

To support energy estimation at various levels of abstraction, other design entities, specified as modules in SystemC, can also be annotated with energy information and appropriate methods can be added to the module definition for communication between hierarchical entities.

6. Conclusions

The intense requirements for low power design, forces the designers of system modeling tools to provide energy estimation capabilities at various levels of the design hierarchy. SystemC, a C++ library, which provides a set of modeling constructs similar to a Hardware Description Language, can be easily extended to offer energy estimating functionality. Taking advantage of the object-oriented nature of SystemC, minor modifications to the definition of modules enable the calculation of the dynamic power component due to logic transitions on the nodes of a digital circuit. The ease in performing these modifications, proves the efficiency of SystemC as a common design language between HW and SW designers.

References:

[1] B. Sirpatil, J. M. Baker Jr., J. R. Armstrong, "Using SystemC to Implement Embedded Software", *International HDL Conference and Exhibition*, San Jose, CA, March 11-12, 2002.

[2] J. Gerlach, W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform", *Workshop on System Design Automation*, Rathen, Germany, March 2000.

[3] L. Cai, P. Kritzinger, M. Olivares, D. Gajski, "Top-Down System Level Design Methodology Using SpecC, VCC and SystemC", *Proc. Design Automation & Test in Europe (DATE) Conference*, Paris, France, March 4-8, 2002.

[4] F. Doucet, V. Sinha and R. Gupta, "Structural Design Composition for C++ Hardware Models", *Proc. Computer Society VLSI Workshop*, 2001.

[5] EmbeddedC++, http://www.caravan.net/ec2plus/

[6] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, 1999.

[7] G. Martin, "UML for Embedded Systems Specification and Design: Motivation and Overview", *Proc. Design Automation & Test in Europe (DATE) Conference*, Paris, France, March 4-8, 2002.

[8] A. Chandrakasan, and R. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic Publishers, Boston, 1995.

[9] J. M. Rabaey, *Digital Integrated Circuits: A Design Perpective,* Prentice Hall, Upper Saddle River, NJ, 1996.

[10] S. Swan, *An Introduction to System Level Modeling in SystemC 2.0*, May 2001, http://www.systemc.org

[11] S. Swan et al., *Functional Specification for SystemC 2.0*, Update for SystemC 2.0.1, April 5, 2002, http://www.systemc.org

[12] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, Boston MA, 1995.