CAD BASED DESIGN COURSE USING A STATE OF THE ART SYSTEM LEVEL LANGUAGE

Suryaprasad Jayadevappa
(esj002c@motorola.com)
Ravi Shankar
(ravi @cse.fau.edu)

1.0 Introduction

Most major U.S universities offer a design course based on Verilog at the undergraduate level. Verilog is used in the high-tech industry to design and develop their commercial products. The increase in design complexity, shortened time to market and intellectual property based methodologies has created a knowledge gap for both the practicing engineer and the new graduate. Today, there is need for higher levels of abstraction and use of system level description languages.

The technology roadmap from the semiconductor industry and a Dataquest market analysis of the EDA (engineering design automation) industry shows that the primary growth in the EDA industry will come from ESL (electronic system level) tools. Similar to the digital design tools of the 1990s, the current and future ESL tools will drive the job market in the SoC (system-on-a-chip) domain over the next decade. A major contender for a unifying language at this level is SystemC. SystemC is based on the C++ language and has constructs to support hardware modeling. The language supports multiple levels of abstraction, a common environment for design and verification, and hardware-software co-design. Currently the SystemC language is undergoing standardization, but has already been adopted by over one hundred design companies. The infrastructure requirement is quiet low as SystemC is open source. Visual C++ and Open source OSCI simulator provide sufficient support to develop SystemC code.

We have developed a CAD Based Computer Design course using SystemC. Thirty five students enrolled in the course that was offered recently. The major challenges in delivering this course were the ability to express hardware components using a high level language preferred for software development and the adaptability of the students. Important SystemC concepts related to hardware modeling was discussed initially. Many design examples developed helped in explaining the concepts and bring out the difference between sequential and concurrent modeling. All the enrolled students had taken a basic course on C++ earlier. In our experience, previous knowledge of C++ helped regarding the syntax, but at times it turned out to have a negative effect. The negative effect was more due to the sequential nature of software programs. We developed a template which is being extensively used for expressing all our designs during the course. It has helped us in sharing our design ideas better. Reusability of the designed models is another important feature that is being stressed upon in this course.

In this paper, we will present our experiences in developing a software-hardware co-design environment, using SystemC, a new concurrent design language. Exposure to this state of the art system level description language and methodology will make the students be better prepared to face the real world industry challenges upon graduation. Armed with the right skill set, the graduates will be productive straight away with very little need for further technical training. The stress on reusability in the design course will also help in visualizing larger SoC based designs built using intellectual property blocks.

In the following sections we discuss in detail the course content and evaluation scheme followed in this course.

2.0 Course Content

At a broad level the course content included an introduction to SystemC and various constructs of SystemC, designing combination and sequential circuits using SystemC, implementing simple finite state machines using SystemC, and finally designing a simple instruction set computer using SystemC.

SystemC can be visualized to consist of a layered architecture as shown in Fig 1 [1]. SystemC being an extension of C++ has additional classes available to support hardware designs. As a pre-requisite for taking this course it was indicated that students have preliminary knowledge of C++ language.

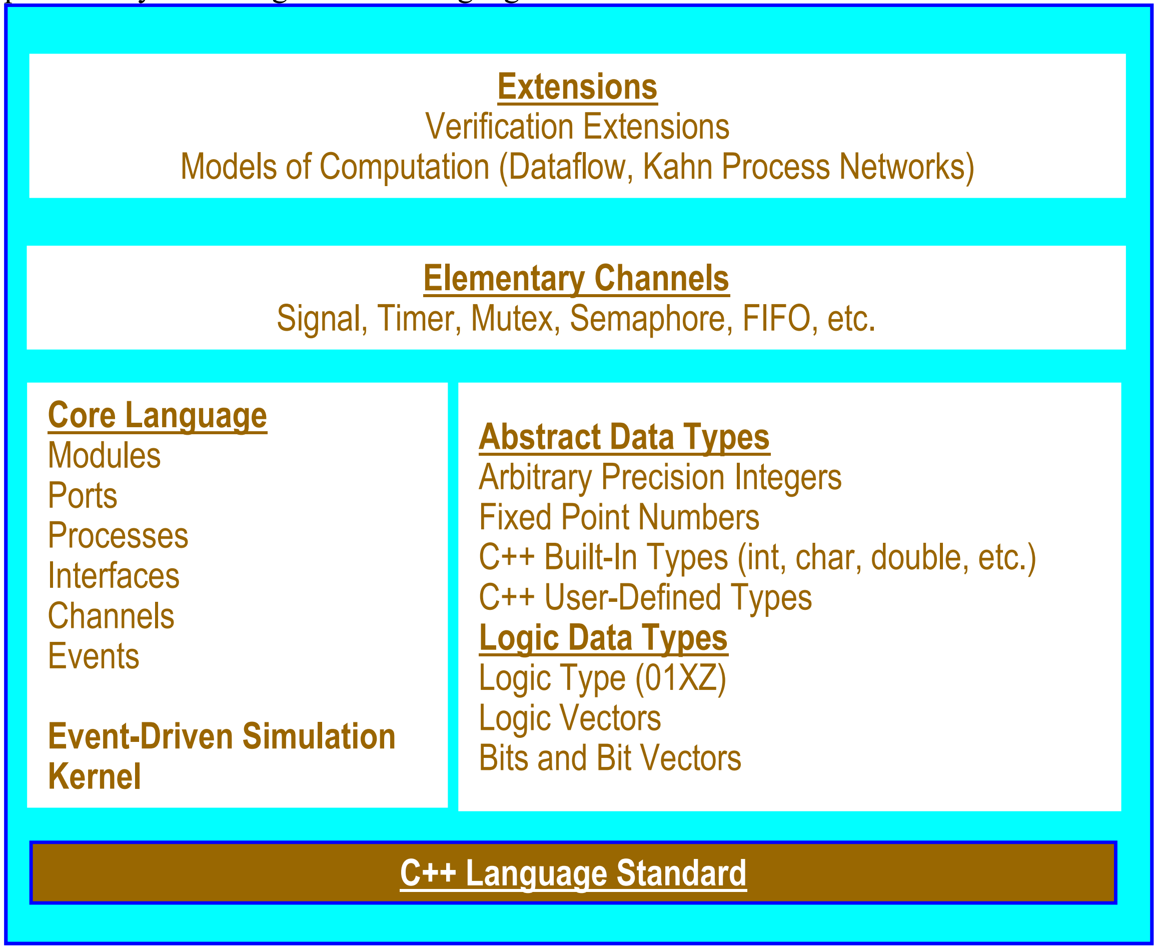


Figure 1. SystemC architecture.

The initial introduction in the course involved the history of SystemC, role of SystemC and the infrastructure requirement and setup for running and simulating SystemC design. Important data types supported in SystemC for expressing hardware design is discussed next. The data types discussed included sc_bit, sc_logic, bit-vector, logic vector, sc_int<>, sc_uint<> and sc_bigint<>. Suitable examples were provided for better understanding of the salient properties of each data type and when to choose one over the other. We also discussed differences between Verilog a hardware description language and SystemC as appropriate.

During the introduction of the core language concepts we used many design examples. Each of the design examples was carefully chosen so as to explain the SystemC core concepts clearly. The design examples varied in complexity starting with the design of a simple 1-bit half adder to more complex designs involving the design of state-machines. All the SystemC code for different designs developed followed a similar pattern as shown in Figure 2. The lowest levels (level 2 and higher) involved the modules providing the functionality of the target design. This could be made up of more layers depending on the complexity and modularity required in the design. Above the module description layer is the "TOP" module (level 1) in which we provide the test bench code for testing the design developed. Finally the sc_main ( ) (level 0) function is from where the simulation begins. Due to the inherent hierarchy followed in this standard pattern it made it easy to develop design models, understand, extend and reuse.

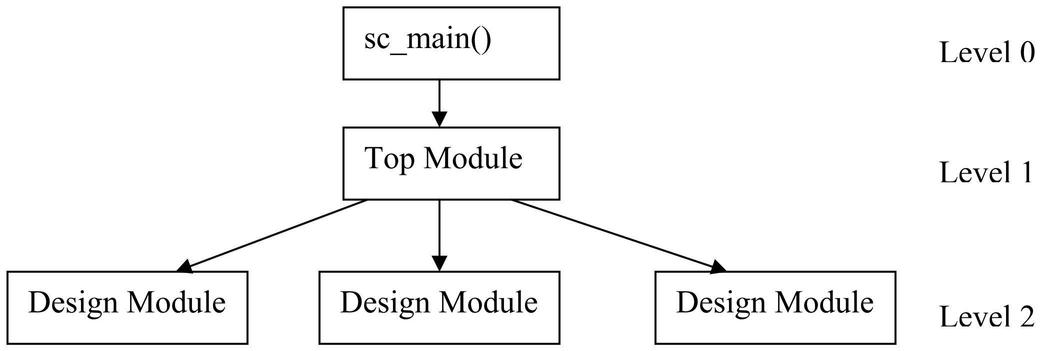


Figure 2. SystemC design module pattern followed.

Reusability is ensured by suitably using the modules developed at lower levels in subsequent higher levels. Reusability in SystemC is also supported by suitable parameterization of the design module. Reusability is showcased with the help of simple example wherein we initially developed a simple 1-bit half adder and use this to develop a 4-bit adder. The example is discussed in more detail here. Figure 3 provides the diagrammatic representation of the 1-bit adder module. Two input ports A and B each of 1-bit size are provided. SUM and Carry are the output ports each of 1 bit size.

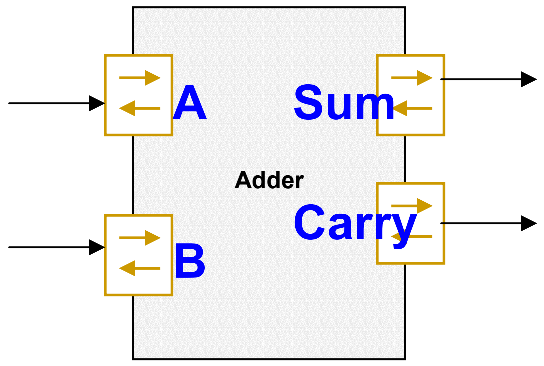


Figure 3. 1-bit Adder Block Diagram.

Figure 4 provides the SystemC code snapshot of the sc_main() or level 0 module where execution begins. In the sc_main() we instantiate the "TOP" module and start the simulation. We control the duration of simulation by providing suitable values for sc_start(). Tracing constructs are also included in the sc_main() to enable waveform traces of signals of interest.

```
int sc_main (int argc , char *argv[])
{
TOP top1("Top1");
sc_start(100, SC_NS);
return 0;  } // End of sc  main
```

Figure 4. 1-bit adder sc_main() – level 0.

In the level 1 or "TOP" module we instantiate the 1-bit adder module or the module designed and perform port binding. SystemC supports both named and positional techniques for port binding. We used the named technique as that provides us flexibility in the ordering of the ports. Also present in the "TOP" module is the test bench which is used to verify the correctness of the module designed. The SystemC code snapshot of the "TOP" module is provided in Figure 5.

```
SC_MODULE(TOP)  { public:  //Various Signals declarations
          sc_signal<sc_bit> in1;          sc_signal<sc_bit> in2;
          sc_signal<sc_bit> sum_out;   sc_signal<sc_bit> carry_out;
          Adder *Adder1;
          SC_HAS_PROCESS(TOP);
          TOP(sc_module_name name) : sc_module(name)    {
                    Adder1 = new Adder("Adder1");
                    Adder1->A(in1);
                    Adder1->B(in2);
                    Adder1->sum(sum_out);
                    Adder1->carry(carry_out);
                    SC_THREAD(main_action);           }
void main_action()
 {   while(1)    {
    in1 =sc_bit ('0');  in2 =sc_bit ('0'); print_out();  wait(10,SC_NS);
    in1 = sc_bit('0'); in2 = sc_bit ('1'); print_out();  wait(10, SC_NS);
    in1 = sc_bit('1'); in2 = sc_bit ('0'); print_out();  wait(10,SC_NS);
    in1 = sc_bit('1'); in2 = sc_bit ('1'); print_out();  wait(10, SC_NS);
   } // END of WHILE LOOP
} //END of main_action
           }; //END of TOP MODULE
```

Figure 5. 1-bit adder TOP module – level 1.

The actual functionality of the 1-bit adder is provided in layer 2. In Figure 6 we present the actual SystemC code implementing the 1-bit adder functionality. The different input and output ports are declared along with their data types. The sensitivity triggering the modules is attached to the processes describing the module.

```
SC_MODULE(Adder)  {
public:  // PORT declarations
          sc_in<sc_bit> A;       sc_in<sc_bit> B;
          sc_out<sc_bit> sum;
          sc_out<sc_bit> carry; // End of Port declarations
          SC_HAS_PROCESS(Adder);
          Adder(sc_module_name name) : sc_module(name)
          {
                    SC_METHOD(main_action);
                    sensitive<<A<<B;     }
void main_action()                       {
          sum = A.read() ^ B.read();
          carry = A.read() & B.read();   }
}; //END OF MODULE
```

Figure 6. 1-Bit Adder SystemC functional description.

To develop a 4-bit adder using the above 1-bit adder, one more layer will be added in between level 1 and level 2 of Figure 2. The lowest layer would consist of the reusable module which is the functional description of the 1 bit adder.

```
SC_MODULE(Adder4bit)  { public: //Various PORT declarations
                sc_in<sc_bv<4> >inputA; sc_in<sc_bv<4> >inputB;
                sc_out<sc_bv<4> >Sum;  sc_out<sc_bit> Carry_out;
        Adder *Adder1, *Adder2, *Adder3, *Adder4;
        SC_HAS_PROCESS(Adder4bit);
        Adder2bit(sc_module_name name) : sc_module(name)
        {       Adder1 = new Adder("Adder1");
                Adder1->A(in1); Adder1->B(in2); Adder1->Carry_in(Carry_in);
                 Adder1->sum(sum_out0); Adder1->carry(temp);
                Adder2 = new Adder("Adder2");
                Adder2->A(in11); Adder2->B(in22);Adder2->Carry_in(temp);
                Adder2->sum(sum_out1);    Adder2->carry(c_out);
                …
                SC_METHOD(main_action);
                sensitive<<inputA<<inputB<<Sum<<Carry_out;    }

}; //END of MODULE
```

Figure 7. SystemC code snapshot of 4-bit adder using 1-bit adder.

The above simple example of 1-bit adder module and its usage to develop a 4-bit adder module introduced the students with various SystemC constructs. Important among them are module, ports, processes, sensitivity and the skeleton of a module description in SystemC. Reusability and a simple way of achieving it using SystemC are also presented.

The next example design presents a sequence detector module developed in SystemC. The sequence detector module developed detects a continous series of three 1's appearing in the input. The sequence detector module is synchronized to a clock signal. This example showcases the ability to develop and simulate sequential circuits. Figure 8 presents the block diagram of the sequence detector module. It has two input ports named Input and Clock, and one output port named Output.

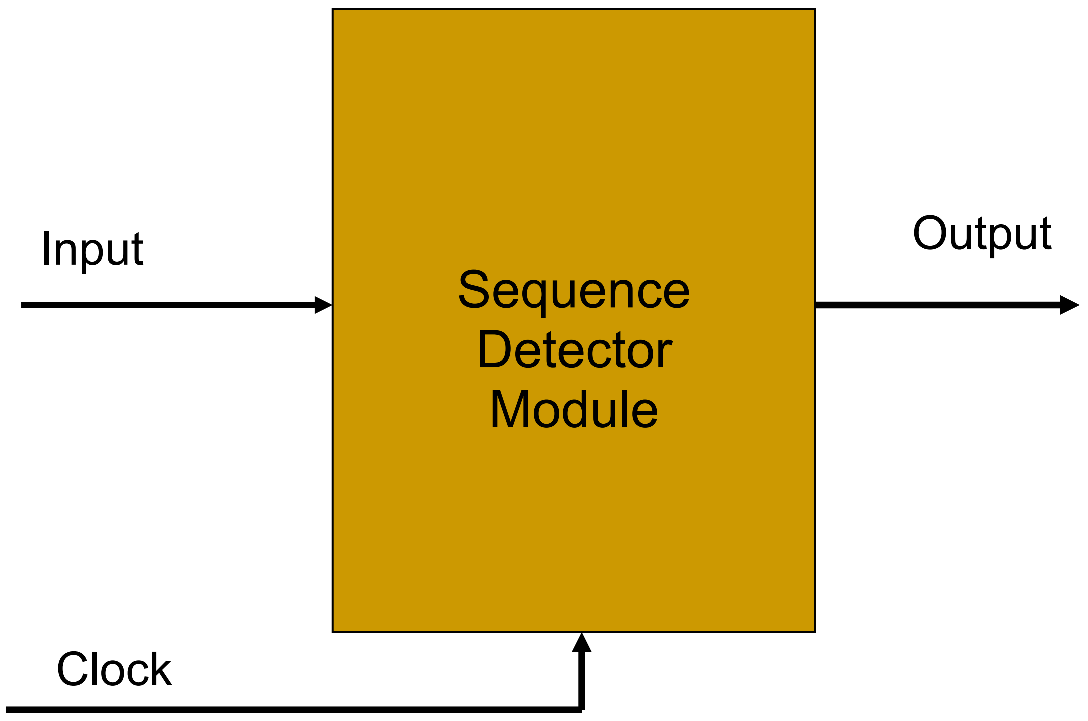


Figure 8. Block diagram of Sequence Detector Module.

Figure 9 presents the digital logic equivalent of the sequence detector module. It is developed using three flip-flops and one three input AND gate. This helped in bridging the gap for the students between the SystemC description and a possible digital logic equivalent. Currently there are no commercially available tools which would generate the gate level equivalent of the SystemC code description.

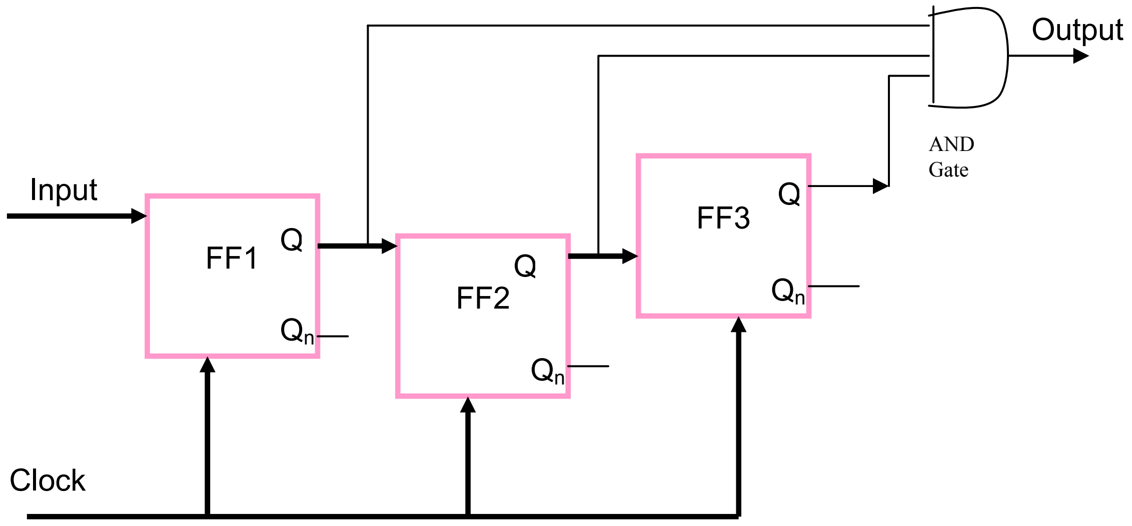


Figure 9. Digital logic equivalent of the Sequence Detector Module.

Figure 10 presents the SystemC code developed for expressing the sequence code detector module.

```
SC_MODULE (detector)
{        //Input Output PORTS
         sc_in<bool> clk, input;
         sc_out<bool > output;
         //Internal Signals
         sc_signal<bool> in1,in2, in3;
         SC_HAS_PROCESS(detector);
                  detector(sc_module_name name) : sc_module(name)
         {
                  SC_METHOD(main_action); //Sequential Logic
                  sensitive_pos<<clk;
                  SC_METHOD(out);        //Combinational Logic
                  sensitive<<in1<<in2<<in3;
         }
void main_action() {
                  in1 = input;
                  in2 = in1;
                  in3 = in2;
                       }
void out()   {
         output = in1 & in2 & in3;
              }
}; // END of DETECTOR module
```

Figure 10. Sequence Detector module functional description in SystemC.

Figure 11 provides the SystemC code snapshot of the TOP (level 1) module. As can been seen, it has structural similarities to that of the 1-bit adder module.

```
SC_MODULE(TOP) {
        public:
        sc_signal<bool> input;
        sc_signal<bool> output;
        sc_clock clk;

        detector *detect;
        SC_HAS_PROCESS(TOP);
        TOP(sc_module_name name) : sc_module(name),clk("clk",10,SC_NS)
        {       detect = new detector("Seq_detector");
                detect->clk(clk);
                detect->input(input);
                detect->output(output);
                SC_THREAD(main_action);
        }
void main_action(){
                while(1)        {
                        input = 1;  print_out(); wait(10,SC_NS);
                        input = 1; print_out(); wait(10,SC_NS);
                                input = 1; print_out();wait(10, SC_NS);
                        input = 0;print_out();wait(10,SC_NS);
                        input = 0;print_out();wait(10,SC_NS);
                        input = 1;print_out();wait(10, SC_NS);
                         } // End of WHILE
                 } // End of main_action
….
}; // END of TOP Module
```

Figure 11. SystemC code snapshot of TOP module for sequence detector.

Figure 12 presents the sc_main() SystemC code for the sequence detector module. It contains additional code used for tracing the various signals of the sequence detector module.

```
int sc_main(int argc, char *argv[])
{    sc_trace_file *tf = sc_create_vcd_trace_file("vcddump");
     TOP top1("top1");
     sc_trace(tf, top1.clk, "CLOCK");
     sc_trace(tf, top1.input, "INPUT");
     sc_trace(tf, top1.output, "OUTPUT");
     sc_start(300,SC_NS);
      sc_close_vcd_trace_file(tf);
    return 0;          }
```

Figure 12. SystemC code of sc_main() for Sequence Detector module.

Figure 13 provides the waveform trace obtained after simulating the sequence detector module. All the input and output ports present in the sequence detector module are traced.

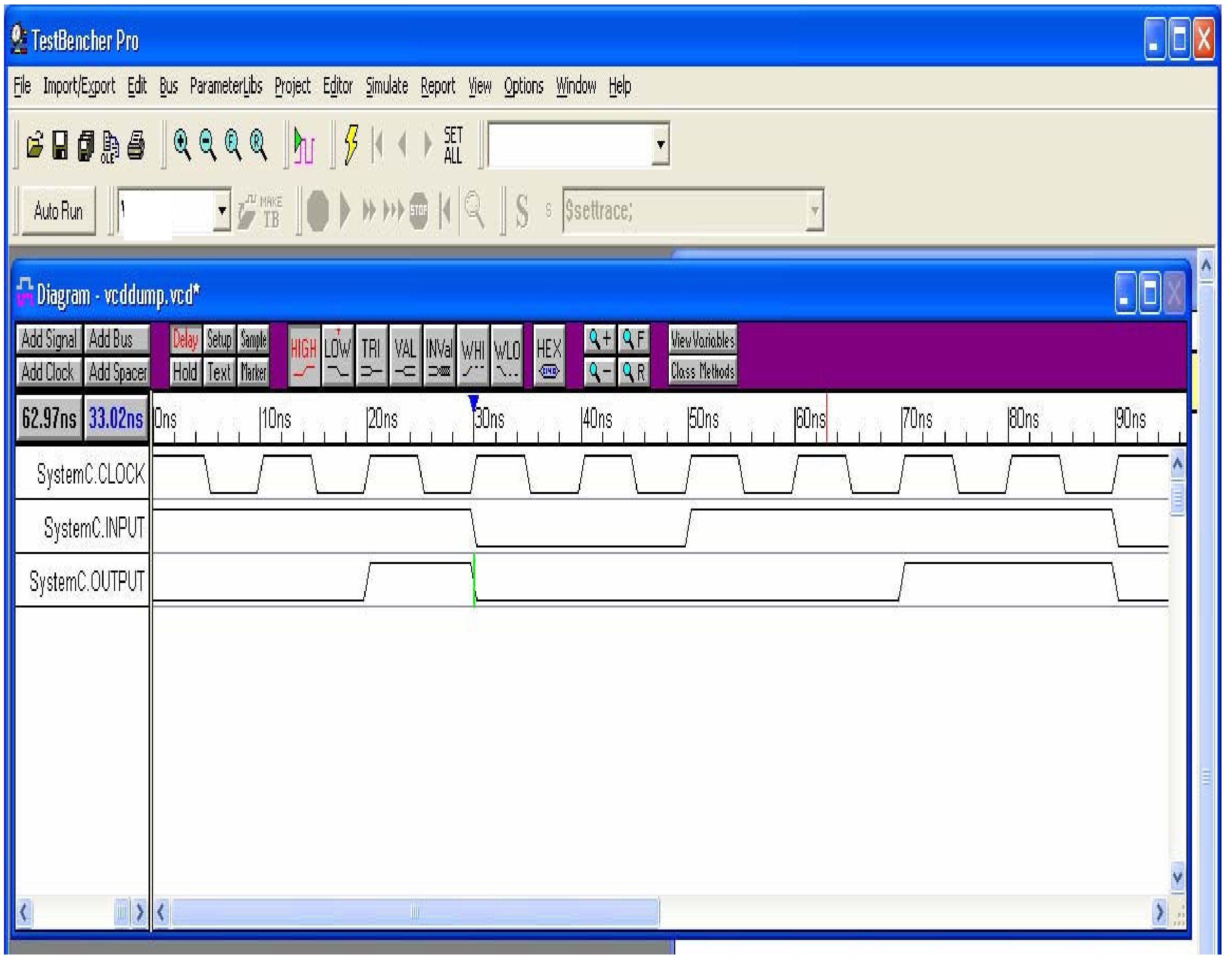

TestBencher Pro
File Import/Export Edit Bus ParameterLibs Project Editor Simulate Report View Options Window Help
Auto Run
$settrace;
Diagram - vcddump.vcd*
Add Signal Add Bus Add Clock Add Spacer
Delay Setup Sample Hold Text Marker
HIGH LOW TRI VAL INVal WHI WLO HEX
View Variables Class Methods
62.97ns 33.02ns
0ns 10ns 20ns 30ns 40ns 50ns 60ns 70ns 80ns 90ns
SystemC.CLOCK
SystemC.INPUT
SystemC.OUTPUT

Figure 13. Sequence Detector Waveform Trace.

Around the seventh week of the course we started with the design of a simple instruction set computer (SISC). A brief introduction of the important concepts in processor design including the fetch, decode, and execute stage was discussed. Figure 14 provides the instruction set and format used in the design of the simple instruction set computer. The instruction set is very similar to the one described by Sternheim, et, al [2].

Instruction Set:
- NOP
- MOVE
- ADD
- SUB
- LOAD
- STORE
- HALT

Instruction Format:

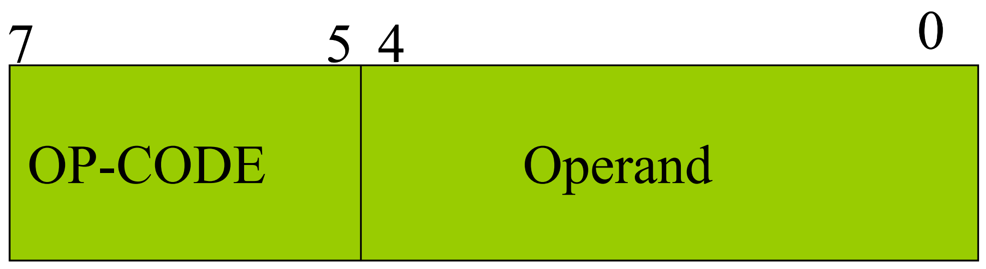


Figure 14. Instruction Set and Format of SISC.

Figure 15 provides a high level view of the SISC and memory module interaction using various ports. After discussing the various registers and their role in the SISC design we first started with the design of the memory module and the test bench code loaded in it.

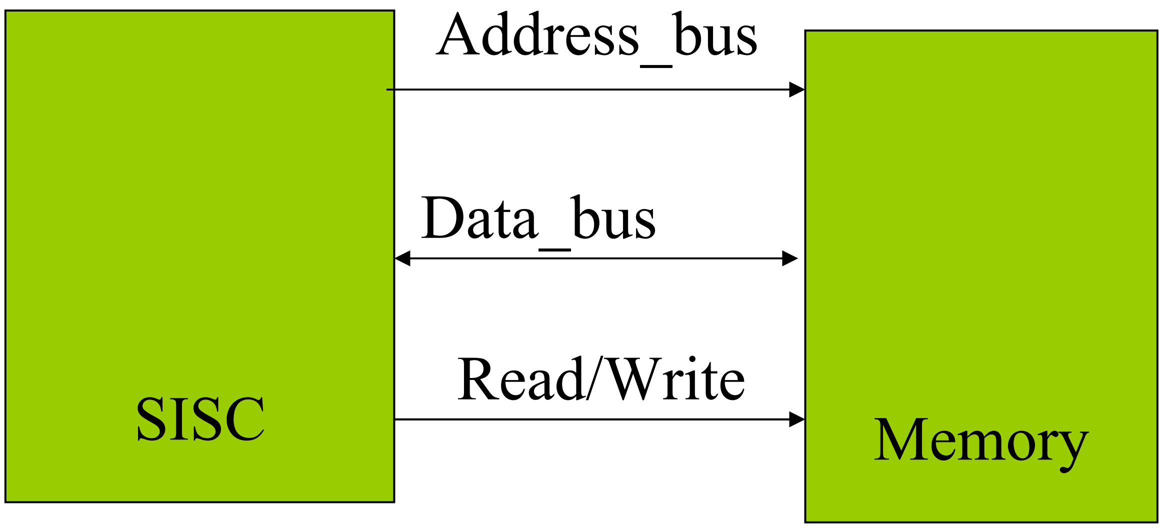


Figure 15. SISC and Memory Module Interaction via various ports.

Figure 16 provides a snapshot of the SystemC code of the Memory module. The memory module was made sensitive to negative edge of the clock.

```
SC_MODULE (Memory)
{       //PORT declarations
          sc_in<bool> rd_wr, clk;
          sc_in<sc_uint<ADDRESS_SIZE> > address_bus;
          sc_inout<sc_bv<DATA_SIZE> > data_bus;

          sc_bv<DATA_SIZE> mem[SIZE];
          SC_HAS_PROCESS(Memory);
          Memory(sc_module_name name):sc_module
          {
                    SC_METHOD(main_action);
                    sensitive_neg<< clk;
          }
};// END of MEMORY module
```

Figure 16. Memory Module SystemC code snapshot.

The rest of the SISC design used similar template as discussed earlier. Figure 17 provides snapshot the sc_main () for the SISC SystemC code. It involved instantiating the Memory and SISC modules and providing suitable port binding. We also traced various signals of interest to verify the cycle accuracy of the design. The TOP module is absent in this design as the test bench code being a part of memory module.

```
int sc_main(int argc, char *argv[])
{
   sc_trace_file *tf = sc_create_vcd_trace_file("vcddump");
          sc_signal<sc_uint<ADDRS_SIZE> > address_bus;
          sc_signal<sc_logic> rd_wr;
          sc_signal<sc_uint<WIDTH> > data_bus;
          sc_clock clk("clk",10, SC_NS); // SISC clock
          sisc1 sisc("sisc");  // Module Instantiation
          memory mem1("mem1");
          //PORT binding
          sisc.address_bus(address_bus);        sisc.rd_wr(rd_wr);
          sisc.data_bus(data_bus);              sisc.clk(clk);
          mem1.address_bus(address_bus);        mem1.rd_wr(rd_wr);
          mem1.data_bus(data_bus);              mem1.clk(clk);
          // Tracing
          sc_trace(tf, sisc.clk, "CLOCK");
         sc_trace(tf, sisc.address_bus, "ADDRESS BUS"); …
          return 0;       } //End of sc_main
```

Figure 17. SystemC code snapshot of sc_main().

Figure 18 provides the SystemC snapshot code of the SISC module. This contained the actual SystemC code necessary for the implementation of the SISC module. The fetch, decode, and execute stages of the processor is very clearly implemented in the design. Modularity is maintained by keeping the actions at each stage separate.

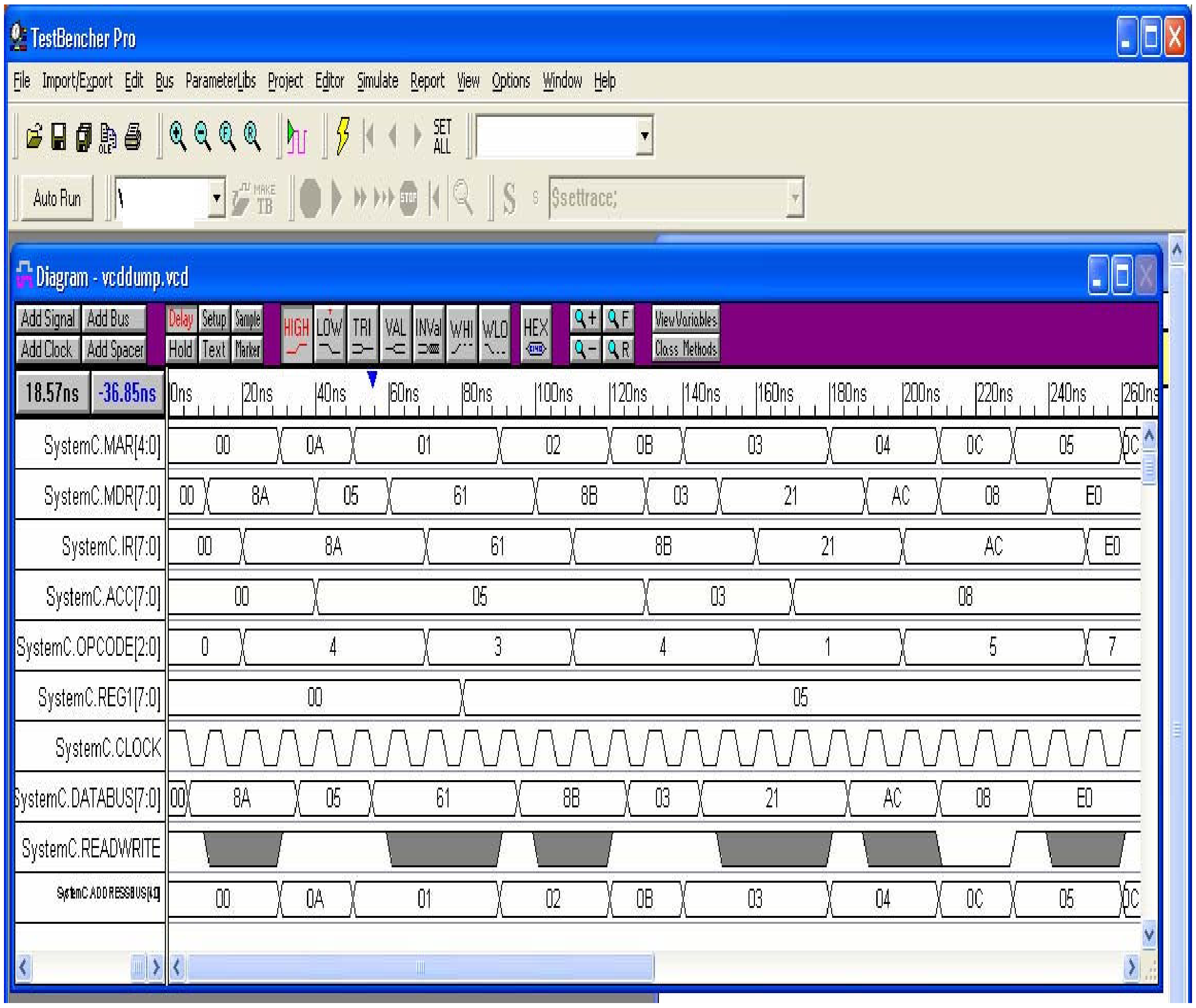
```
SC_MODULE(sisc1)
{
        sc_out<sc_uint<ADDRS_SIZE> > address_bus;
        sc_out<sc_logic> rd_wr;
        sc_inout<sc_uint<WIDTH> > data_bus;
        sc_in<bool> clk;

        SC_HAS_PROCESS(sisc1);

        sisc1(sc_module_name name):sc_module(name)
        {
                init1();
                SC_THREAD(main_action);
        }
        void main_action()
        {
                while(1)
                {       instr_fetch();   wait(CYCLE, SC_NS);
                        instr_decode();wait(CYCLE, SC_NS);
                        instr_exec();    wait(CYCLE,SC_NS);
                }
        } ….
}; // END of SISC Module
```

Figure 18. SystemC code snapshot of SISC module.

The above SystemC code of the SISC module was extended in the term project by the students by suitably extending the instruction set to include a larger instruction set and supporting different addressing modes. The detailed waveform trace of the SISC module designed is presented in Figure 19. The test bench code developed was simple and performed addition of two numbers N1 and N2. The two numbers N1 and N2 are stored at different memory locations and the result was stored in another memory location N3.


TestBencher Pro
File Import/Export Edit Bus ParameterLibs Project Editor Simulate Report View Options Window Help
Auto Run
$settrace;
Diagram - vcddump.vcd
Add Signal Add Bus Add Clock Add Spacer
Delay Setup Sample Hold Text Marker
HIGH LOW TRI VAL INVal WHI WLO HEX
View Variables Class Methods
18.57ns -36.85ns
0ns 20ns 40ns 60ns 80ns 100ns 120ns 140ns 160ns 180ns 200ns 220ns 240ns 260ns
SystemC.MAR[4:0]
SystemC.MDR[7:0]
SystemC.IR[7:0]
SystemC.ACC[7:0]
SystemC.OPCODE[2:0]
SystemC.REG1[7:0]
SystemC.CLOCK
SystemC.DATABUS[7:0]
SystemC.READWRITE
SystemC.ADDRESSBUS[4:0]

```
SystemC.MAR[4:0]       00 0A 01 02 0B 03 04 0C 05 0C
SystemC.MDR[7:0]       00 8A 05 61 8B 03 21 AC 08 E0
SystemC.IR[7:0]        00 8A 61 8B 21 AC E0
SystemC.ACC[7:0]       00 05 03 08
SystemC.OPCODE[2:0]    0 4 3 4 1 5 7
SystemC.REG1[7:0]      00 05
SystemC.DATABUS[7:0]   00 8A 05 61 8B 03 21 AC 08 E0
SystemC.ADDRESSBUS[4:0] 00 0A 01 02 0B 03 04 0C 05 0C
```

Figure 19. Detailed Waveform trace of the SISC module.

3. Evaluation

The course was designed to be a hands-on laboratory intense course with three laboratory assignments, two tests and one term project. Some of the laboratory assignments involved extending few of the design examples developed and provided in class. For example, part of the first assignment involved reusing the 1-bit adder to develop a 4-bit adder. The other part of first assignment involved reusing the 4-bit adder to develop a 8 bit adder.

For the term project a simple instruction set computer was first discussed in detail in class. This involved the various stages of the processor design including the SystemC design code being made available. Later the students were given a specific instruction set with few addressing modes and encouraged to develop the design for the processor. To maintain consistency, students used the same instruction bit encoding for the instructions. The test bench used for verifying the processor designed was also standardized. Many students reused majority of the code used for the design of the simple instruction set computer.

In the first test students were evaluated on the understanding of the SystemC concepts. The ability to develop SystemC code for combinational and sequential logic, and involving state machines was tested. The second test was completely based on the processor design. Students were tested on their ability to make suitable changes to their term project to accommodate newer instructions and addressing modes.

4. Observations

Our aim was to expose the students to a state of the art system level design language – SystemC. Developing a new course and handling a class strength of thirty five students was a real challenge. Exposure to courses such as C++, course on logic design, and an introductory course on microprocessors were key to the successful designs in this course. We concentrated more on SystemC concepts and processor design. Currently there is no single book which does both. Knowledge of C++ was helpful in debugging the assignments and term project designs. But it also caused hindrance in understanding of the concurrent support as required in hardware design. This was overcome by a variety of design examples done in class. Also the usages of a standard template in expressing the designs further helped in easier understanding and visualize reusability of design. The "SystemC Primer"[3] and the SystemC [4] website helped as good references for the students. A major advantage is the infrastructure cost involved. As SystemC is available freely, the only cost involved in terms of simulating the designs was that of the cost of an ANSI compliant C++ compiler. Over seventy percent of the participating students owned a computer. Each one had the necessary environment setup for running SystemC designs. This provided flexibility in terms of the laboratory resource requirement and also making it a laboratory intense course with many assignments involving designs. Over 90 % of the participating class was able to perform inline or above the expectation set for the class.

5.References

[1] Grotker, T., Liao, S., Martin, G., and Swan, S., " System Level Design using SystemC", Kluwer Academic Press, 2001.
[2] Sterheim, E., Singh, R., Madhavan, R., Trivedi, Y., "Digital Design and Synthesis with Verilog HDL", Automata Publishing Company 1993.
[3] Bhasker, J., " SystemC Primer", Star Galaxy Publishing, 2002.
[4] SystemC website, www.systemc.org
[5] Arnold, M., G., "Verilog Digital Computer Design – Algorithms to Hardware", Prentice Hall 1999.

Biographical Information

SURYAPRASAD JAYADEVAPPA

Received his PhD in computer engineering from Florida Atlantic University in 2003. He worked in various teaching capacities offering many computer sciences and engineering courses for over 8 years. He worked as a summer intern at Cadence Design Systems in 2001. After which he worked there for over 14 months. Currently he is working with Motorola. His research interests include high level system design methodologies, embedded system design and design automation.

RAVI SHANKAR

Professor Ravi Shankar has a PhD in electrical and computer engineering from the University of Wisconsin, Madison, WI, and an MBA from the Florida Atlantic University (FAU), Boca Raton, FL. He is the director of a college-wide center at FAU on chip and system design. He has been a consultant to Motorola, IBM, and Cadence. At present he is leading a major industrial project on enhancing system design productivity.