SystemC For System Design: Our Academic Experience

Ravi Shankar
Computer Science and Engineering, Florida Atlantic University,

Carlos Krieghoff
Computer Science and Engineering, Florida Atlantic University

---

SystemC 2.0 is a system level modeling and design language based on C++. It provides a single common language to describe software and hardware, and the integration there of. It provides support for various levels of abstraction for system description as well as concurrency, communication, and synchronization. We have used SystemC in a core computer engineering course since fall 2002, entitled CAD-Based Computer Design, which was originally taught with, first VHDL and, then, Verilog HDL. Our goal here has been to attract both computer engineering and computer science majors to use this common platform to interact, communicate, and build useful designs. The course has evolved, along with the evolution of SystemC 2.0, from the RTL level to higher levels of abstraction, and finally has led us to a course on concurrency modeling, being taught this semester.

Categories: D.1 [**Software**]: Programming Techniques – *object oriented programming and concurrent programming*; H.1 [**Information Systems**]: Models and Principles – *Miscellaneous*; I.6 [**Computing Methodologies**]: Simulation and Modeling – *Model Validation and Analysis and Model Development*; and J.6 [**Computer Applications**]: Computer-Aided Engineering – *Computer-aided design*.

General Terms: Human Factors, Management, and Standardization

Additional Key Words and Phrases: SystemC, Software-Hardware CoDesign, and Abstractions.

---

1. Introduction

U.S universities offer a digital/computer design course based on Verilog HDL (hardware description language) or VHDL (VHSIC – very high speed integrated circuits – HDL) at the undergraduate level. They are 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. Such languages would model software and hardware together; address the issues of concurrency, synchronization, and communication; and allow one to perform what-if scenarios at various levels of abstraction.

SystemC is a modeling language that was developed by the Open SystemC Initiative (OSCI) planning group, to address the needs of system level software-hardware codesign. SystemC is based on the C++ language. SystemC 1.0 provides a set of modeling constructs that are similar to those used in RTL (register transfer level) and behavioral modeling within an HDL. SystemC 2.0, released in 2001, improves upon SystemC 1.0, to enable system level modeling – that is, modeling of systems above the RTL level of abstraction, including systems which might be implemented in software, hardware, or some combination of the two. It introduces a new set of features for generalized modeling of communication and synchronization, called channels, interfaces, and events.

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. SystemC is designed with a small general purpose modeling foundation, so that one can easily add various models of computation, design libraries, modeling guidelines, and design methodologies as required for system design [OSCI, 2002].

We detail below our experiences over the past two years with an undergraduate computer engineering core course entitled “CAD-Based Computer Design.” In earlier offerings, the course used, first VHDL, then Verilog HDL, to design, a 16-bit central processing unit. Though the course attracted a few computer science majors on occasion, the job trends and the difficulty in learning a new concurrent language made it primarily a computer engineering course. However, several recent trends persuaded us to rethink the status quo: The number of world-wide ASIC designers is shrinking, thanks to the automation unleashed by the EDA (engineering design automation) industry, and resulting increase in productivity; The technology continues to shrink the transistor and provide more functionality at lower cost, blurring the cost differential between ASICs (application specific integrated circuits) and FPGAs (field programmable gate arrays); Consumers are continuing to demand more complex functionality in their (typically) mobile wireless embedded systems; Increasing system complexity from all these trends has led to significant design reuse and the evolution of many IP (intellectual property) design companies; and Complexity management with a divide-and-conquer approach and 3rd party IP bocks has actually led to system level issues as the major show stoppers for the high tech industry. Thus, we concluded that new productivity challenges (and hence, jobs) are at the system level, specifically with regard to software-hardware co-design, co-verification, and performance tradeoffs (One can add executable specifications to this list, which is beyond the scope of this paper). Further, Dataquest has predicted that the primary growth in the EDA industry will come from ESL (electronic system level) tools [DataQuest, 2002]. Thus, we concluded that 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.

Fortunately for us, two years ago, the competition between SystemC and System Verilog was heating up. It gave us an opportunity to compare two approaches that originated from two distinctly different philosophical perspectives. SystemC proponents wanted both software and hardware practitioners to interact more comfortably, while the System Verilog proponents viewed system design as primarily the forte of the hardware engineer who was already familiar with Verilog and had access to good back-end automation tools. The jury is still out as to which approach will ultimately succeed, since we see pros and cons to both sides of the argument. However, we decided to go the route of SystemC, for a number of reasons: (1) Since it is C++ based, software engineers will accept it, while hardware engineers, at least the recent graduates, will have familiarity with the C++ language. Thus, a common platform for communication existed; (2) One could use low cost tools for simulation. Traditionally, the EDA tools have needed a steep learning curve and a large budget (relative to the cost of software tools) to maintain; and (3) SytemC allows modeling and design at multiple levels of abstraction, with the prospect of supporting software developers with a truer hardware model, system architects with high and mixed level models for their what-if scenarios, and hardware designers with EDA support for automated translation/synthesis [Shankar and Suryaprasad, 2002]. However, SystemC simulations are slow relative to the simulation speed achieved by stand-alone software and hardware development environments. We view this as a current limitation and technologies exist/ will evolve to address this, if indeed SystemC turns out to be a viable medium for system design. As such, our role, as an academic institution, was deemed to explore SystemC fully and thoroughly, and provide feedback on our experiences.

2. Methods

2.1 The Course Outline:

Carlos, Include material from Surya's paper here. Start at 'Course Content' on page 2 and use most of his writeup through the end of page 3.

The majority of the students taking the CAD Based Computer Design course are in their junior or senior year. As a result, they all have C++ programming experience from other required courses such as: Foundations of Computer Science or Data Structures where they become familiar with this particular programming language. The first surprise the students have though is when they learn that we will be modeling hardware using a programming language. They feel motivated. Thus, we begin the CAD Based Computer Design course by exploring the history of hardware-software co-design during the past several years.

Subsequently, we introduce to the students the different SystemC constructs that allow us to model hardware using C++. Such SystemC constructs are new defined data types and concepts like ports and signals, and the declaration and implementation of processes using SC_METHODs or SC_THREADs. The usage of the SystemC constructs is shown to the students through a variety of simple examples (1-bit adder, multiplexer, counter, etc). Figure 1 shows the basic SystemC structure used to simulate systems. The system that is simulated is instantiated with the driver and monitor SystemC modules. The driver will generate the stimulus to the system and the monitor will gather the results. The results can be displayed on the screen, saved on a text file or saved as a waveform format.

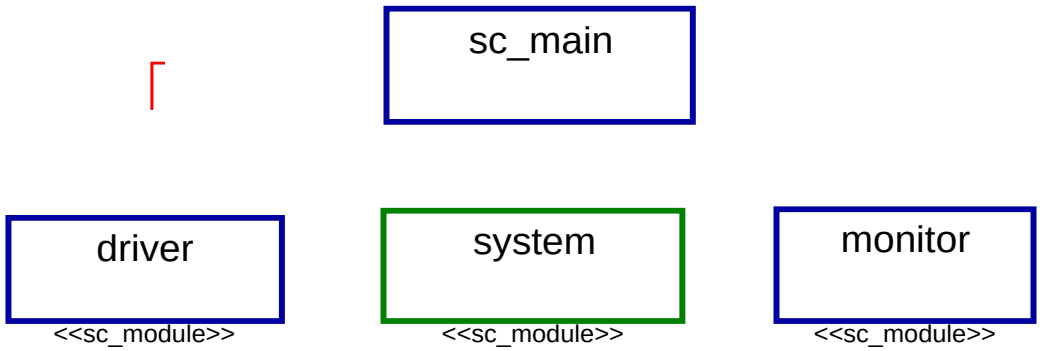


Figure 1 – Basic SystemC modeling class diagram

2.2 Our Course Offerings:

We have taught the "CAD-Based Computer Design" three times, over the past 2 years, using SystemC.

In the first offering, we had a class of 52 seniors from computer science, computer engineering, and electrical engineering undergraduate programs. We taught SystemC and the conventional concepts of hierarchy, modularity, reuse, and state machine design. Since this was our first offering, we kept our designs close to the RTL level as covered in an excellent introductory book [Bhasker, 2001]. However, to inject the perspectives of cooperation and competition, we also involved the class in a half-semester long project that simulated the IP (Intellectual Property) market dynamics. The goal was to provide students with a real-life experience of IP development and usage and provide an entrepreneurial experience while providing credit for their participation. The IP market we simulated had 3 major participants, namely the IP developers, EDA vendors, and System Designers. The IP companies developed basic building blocks for digital design. Their deliverables included IP design models, source code, and application notes. These were then submitted to EDA vendors whose primary objective was to rank these IP on a scale of 1 to 5. EDA vendors were also invited to develop tools to facilitate IP development and integration. System companies chose the better ranked IP blocks to integrate and model their system talk. By self-selection, the class ended up with 16 IP companies, 4 EDA companies, and 6 System companies. The project goal was to build a fixed point (16/32-bit) or floating point (IEEE Standard) ALU (arithmetic and logic unit)

using generic building blocks. Building blocks implemented in SystemC were provided by IP companies, certified by EDA companies, and integrated by System Companies. Our experiences are documented in an earlier publication [Quraishi and Shakar, 2003]. While some students …

In the second offering, we had a class of thirty five students. This was taught by a PhD student in the computer engineering program with a strong background in computer science. Our goal was to elevate the course to system level design at a higher level of abstraction. Important SystemC concepts related to hardware modeling were discussed initially. Many design examples developed helped in explaining the concepts and bring out the difference between sequential and concurrent modeling. During the second half of the semester, design of a simple instruction set computer, as described in an early book on Digital Design with Verilog [Sterhieim, et al., 1993], was adapted to SystemC. We covered a fair amount of the design at the behavioral level in the class. The students were then given a specific instruction set with few addressing modes and asked to develop the design of the processor. The instruction set and the test bench were standardized. Many students reused majority of the code developed in the class.

In our third offering, the course was offered on we extended the course to represent abstraction, both for combination and sequential designs. In addition to the classical CPU design, we also chose the example of a robotic controller to bring out the concepts of concurrency. In the next section, we provide details of all of these examples. ….

3. Projects

3.1 Four Bit-Adder: A Combination Design Example

3.1.1 Four-bit Adder: behavioral level of abstraction

Following a class discussion on a 1-bit adder using half adders, we assigned the students the modeling of a 4-bit adder. The 4-bit adders had to be modeled at three different levels of abstraction: behavioral, Boolean and gate. Figure 2 shows the class diagram of the 4-bit adder system at the behavioral level of abstraction.

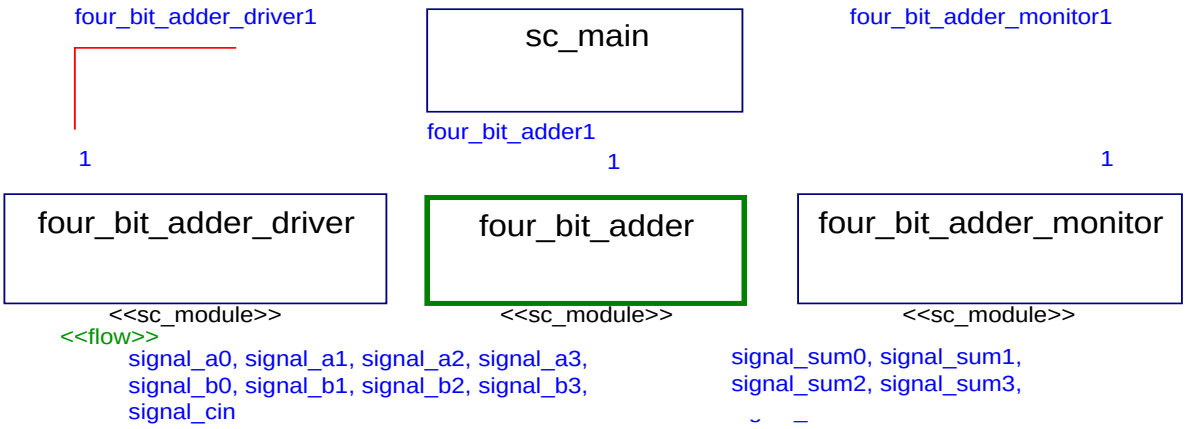


Figure 2 - 4-bit adder top class diagram

As we mentioned earlier, the simulation contains three main modules: the system being simulated, the driver and the monitor. Communication among the SystemC modules are shown using flows. The flows indicate the port bindings of modules, which are performed using SystemC signals. Part of the code from the sc_main below shows the port binding between the driver and the 4-bit adder modules.

```
four_bit_adder_driver four_bit_adder_driver1 ( "four_bit_adder_driver" );

four_bit_adder_driver1.driver_a0 ( signal_a0 );
four_bit_adder_driver1.driver_a1 ( signal_a1 );
four_bit_adder_driver1.driver_a2 ( signal_a2 );
four_bit_adder_driver1.driver_a3 ( signal_a3 );

four_bit_adder_driver1.driver_b0 ( signal_b0 );
four_bit_adder_driver1.driver_b1 ( signal_b1 );
four_bit_adder_driver1.driver_b2 ( signal_b2 );
four_bit_adder_driver1.driver_b3 ( signal_b3 );

four_bit_adder_driver1.driver_cin ( signal_cin );

four_bit_adder four_bit_adder1("four_bit_adder1");

four_bit_adder1.fourBitcarry_in1( signal_cin);

four_bit_adder1.fourBitinA1(signal_a0);
four_bit_adder1.fourBitinA2(signal_a1);
four_bit_adder1.fourBitinA3(signal_a2);
four_bit_adder1.fourBitinA4(signal_a3);
```

```
four_bit_adder1.fourBitinB1(signal_b0);
four_bit_adder1.fourBitinB2(signal_b1);
four_bit_adder1.fourBitinB3(signal_b2);
four_bit_adder1.fourBitinB4(signal_b3);
```

The declaration of the four_bit_adder module and the behavioral implementation of the four_bit_adder is shown below. The process main_action is declared as a SC_METHOD. This process will execute every time there is change in the input ports declared in the sensitivity list. In the behavioral level of abstraction, the addition of the two four bits numbers and the carry is performed in the four_bit_adder module itself.

```
four_bit_adder ( sc_module_name name ) : sc_module ( name )
        {

                SC_METHOD ( main_action );

                sensitive << fourBitinA1 << fourBitinB1 << fourBitcarry_in1;
                sensitive << fourBitinA2 << fourBitinB2;
                sensitive << fourBitinA3 << fourBitinB3;
                sensitive << fourBitinA4 << fourBitinB4;

}

void four_bit_adder::main_action()
{
                vec4_inA[0] = sc_bit(fourBitinA1.read());
                vec4_inA[1] = sc_bit(fourBitinA2.read());
                vec4_inA[2] = sc_bit(fourBitinA3.read());
                vec4_inA[3] = sc_bit(fourBitinA4.read());

                vec4_inB[0] = sc_bit(fourBitinB1.read());
                vec4_inB[1] = sc_bit(fourBitinB2.read());
                vec4_inB[2] = sc_bit(fourBitinB3.read());
                vec4_inB[3] = sc_bit(fourBitinB4.read());

                vec4_sum = vec4_inA + vec4_inB + sc_bit(fourBitcarry_in1.read());

                fourBitsum_out1.write((sc_bit)vec4_sum[0]);
                fourBitsum_out2.write((sc_bit)vec4_sum[1]);
                fourBitsum_out3.write((sc_bit)vec4_sum[2]);
                fourBitsum_out4.write((sc_bit)vec4_sum[3]);

                fourBitcarry_out4.write((sc_bit)vec4_sum[4]);

}// End of main_action
```

Figure 3 shows the result of the simulation.

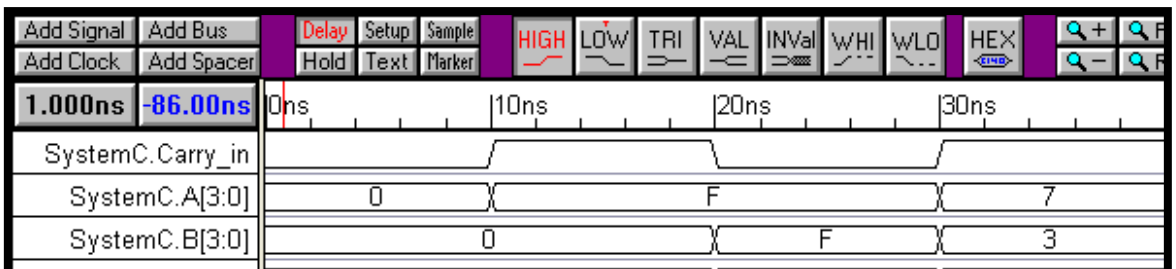


Figure 3 – Output waveform: 4-bit adder at behavioral level

3.1.2 Four-bit Adder: boolean level of abstraction

In the boolean implementation of the 4-bit adder, the four_bit_adder module instantiates four full_adder modules. Then, the full_adder modules are themselves instantiate two half_adder modules. Figure 4 and figure 5 show the second and third level class diagrams of the 4-bit adder system. The top level class diagram of the Boolean level is the same as class diagram shown in figure 2.

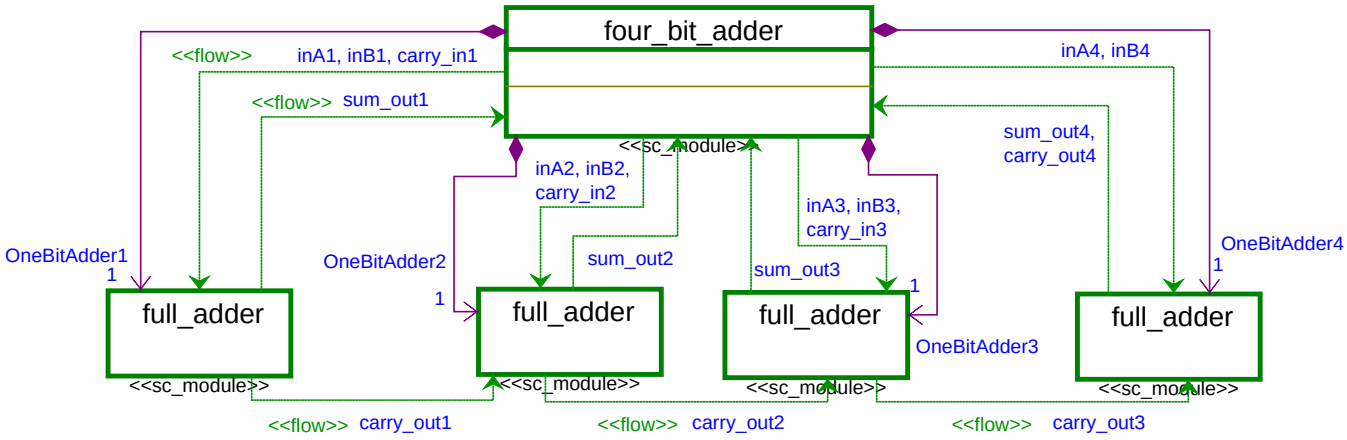


Figure 4 – 4-bit adder at Boolean level: second level class diagram

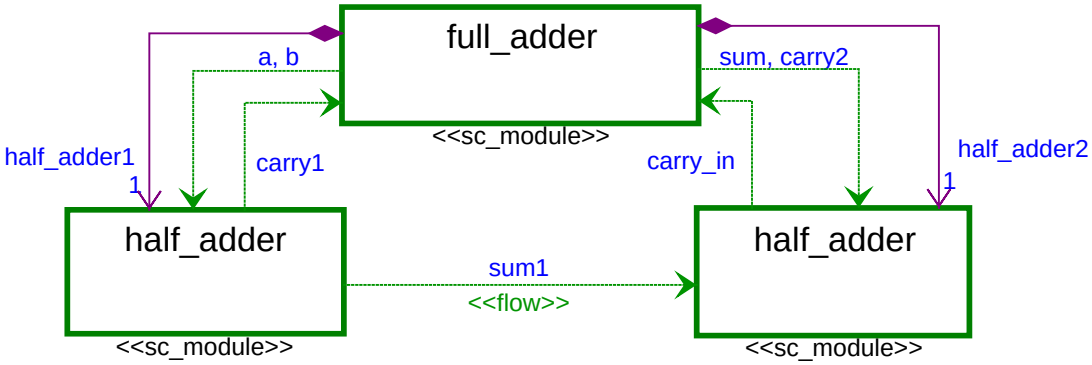


Figure 5 – 4-bit adder at Boolean level: third level class diagram

The code below shows the declaration of the full_adder module as described in the class diagram above.

```
SC_HAS_PROCESS ( full_adder );
        full_adder ( sc_module_name name ) : sc_module ( name )
        {
                half_adder1 = new half_adder ( "ha1" );  // Creating pointer to module + port
bindings
                half_adder1->a ( a );                                   // for first half adder module
                half_adder1->b ( b );
                half_adder1->sum ( sum1 );
                half_adder1->carry_out ( carry1 );

                half_adder2 = new half_adder ( "ha2" );  // Creating pointer to module + port
bindings
                half_adder2->a ( sum1 );                                        // for second half adder
module
                half_adder2->b ( carry_in );
                half_adder2->sum ( sum );
                half_adder2->carry_out ( carry2 );

                SC_METHOD ( main_action );
                sensitive << carry1 << carry2; //Sensitivity list
        }
```

This is the implementation of the full_adder, which is calculating the carry of the sum of two bits.

```
void full_adder::main_action ( )
{
        carry_out.write(carry1.read() | carry2.read());
}
```

This is the implementation of the half_adder, which is calculating the sum of two bits.

```
void half_adder::main_action ( )
{
        sum.write(a.read() ^ b.read());
        carry_out.write(a.read() & b.read());

}
```

Figure 6 shows the result of the simulation.

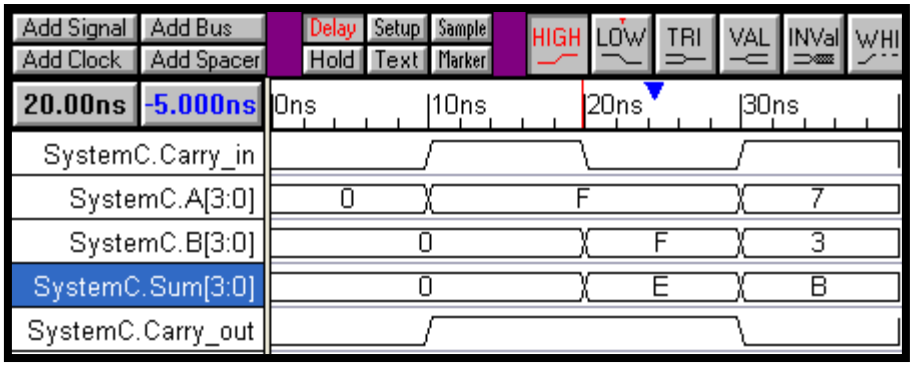


Figure 6 – Output waveform: 4-bit adder at boolean level

3.1.3 Four-bit Adder: gate level of abstraction

In the four-bit adder at the gate level of abstraction, we add another hierarchy below the half_adder module. In this case, the implementation of the half_adder module is performed by NAND gates. Figure 7 shows the fourth class diagram of the four-bit adder at the gate level of abstraction.

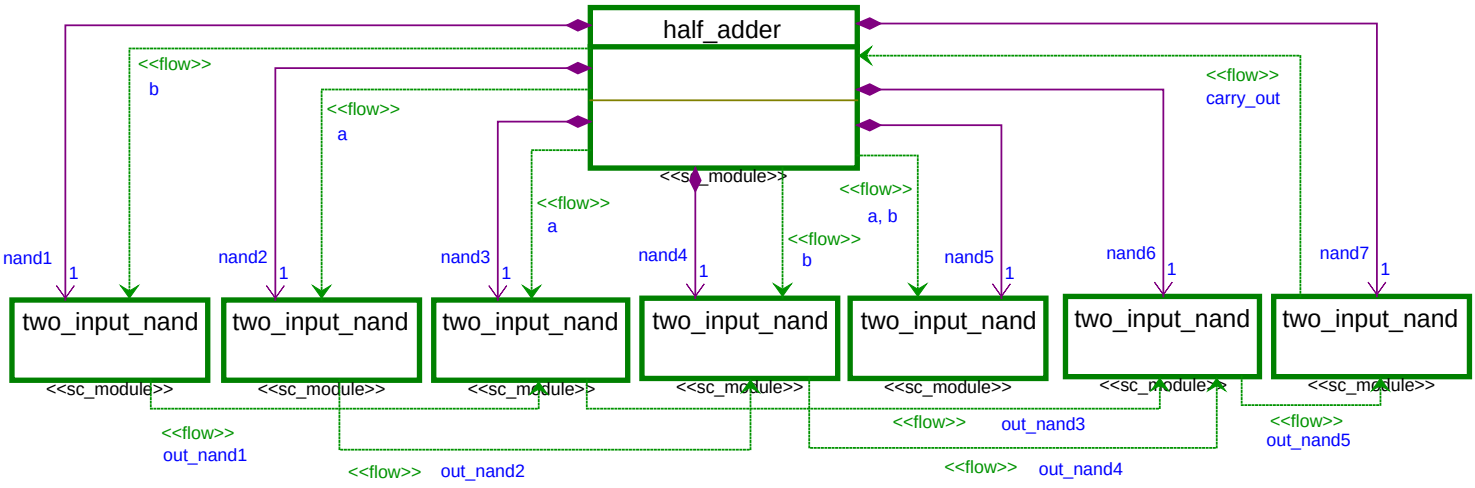


Figure 7 – 4-bit adder at Boolean level: fourth level class diagram

The instantiation of the NAND gates is below.

```
SC_HAS_PROCESS ( half_adder );
        half_adder ( sc_module_name name ) : sc_module ( name )
        {

                nand1_ptr = new two_input_nand ( "nand1" );
                nand1_ptr->a ( b );
                nand1_ptr->b ( b );
                nand1_ptr->out_nand ( signal_out_nannd1 );

                nand2_ptr = new two_input_nand ( "nand2" );
                nand2_ptr->a ( a );
                nand2_ptr->b ( a );
                nand2_ptr->out_nand ( signal_out_nannd2 );

                nand3_ptr = new two_input_nand ( "nand3" );
                nand3_ptr->a ( a );
                nand3_ptr->b ( signal_out_nannd1 );
                nand3_ptr->out_nand ( signal_out_nannd3 );

                nand4_ptr = new two_input_nand ( "nand4" );
                nand4_ptr->a ( signal_out_nannd2 );
                nand4_ptr->b ( b );
                nand4_ptr->out_nand ( signal_out_nannd4 );

                nand5_ptr = new two_input_nand ( "nand5" );
                nand5_ptr->a ( a );
                nand5_ptr->b ( b );
                nand5_ptr->out_nand ( signal_out_nannd5 );

                nand6_ptr = new two_input_nand ( "nand6" );
                nand6_ptr->a ( signal_out_nannd3 );
                nand6_ptr->b ( signal_out_nannd4 );
                nand6_ptr->out_nand ( sum );

                nand7_ptr = new two_input_nand ( "nand7" );
                nand7_ptr->a ( signal_out_nannd5 );
                nand7_ptr->b ( signal_out_nannd5 );
                nand7_ptr->out_nand ( carry_out );

        }
};
```

The process below belongs to the implementation of the two_input_nand module.

```
void two_input_nand::prc_two_input_nand()
{
        out_nand.write( ~(a.read() & b.read()) );
}
```

Figure 8 shows the result of the simulation.

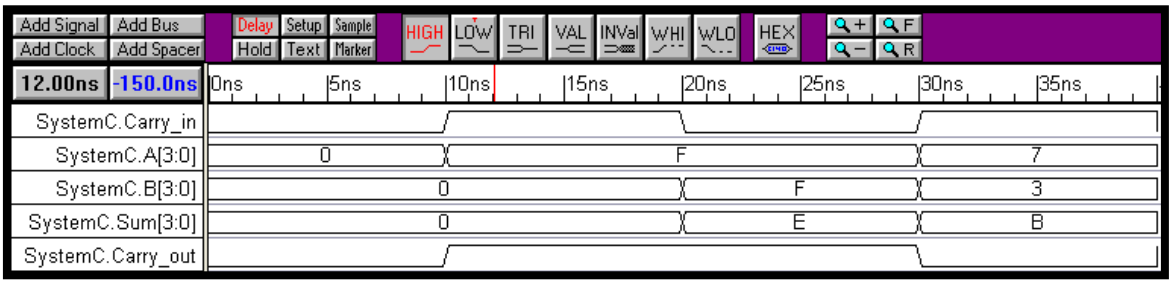


Figure 8 – Output waveform: 4-bit adder at gate level

3.2 Sequence Detector: A Sequential Design Example

3.2.1 Sequence detector: behavioral level of abstraction

The model of the sequence detector at the behavioral level of abstraction contains three main modules: the system being simulated, which in this case is the sequence detector, the driver and the monitor. Communication among the SystemC modules are shown using flows, such is the case of the enable and data signal transmitted from the driver to the sequence detector module and the valid_out and the seq_found signals going from the sequence detector module to the monitor module.

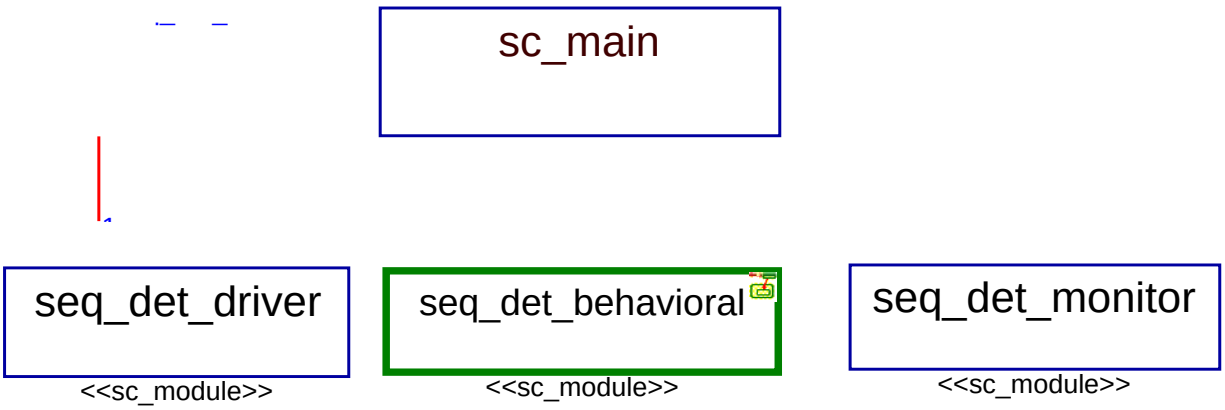


Figure 9 – Sequence detector at behavioral level: top level class diagram

The following code shows the declaration of the main process of the sequence_detector module.

```
SC_HAS_PROCESS(seq_det_behavioral);
seq_det_behavioral(sc_module_name name) : sc_module(name)
{
        SC_THREAD(main_action);
}
```

The main process of the sequence detector in the behavioral level of abstraction is a thread that is in charge of storing the incoming bits and comparing them to verify if the data is valid and if the sequence was found.

```
void seq_det_behavioral::main_action()
{
        while(1)
        {

                if(enable.read()==sc_logic_1)
                {
                        data_array[2] = data_array[1];
                        data_array[1] = data_array[0];
                        data_array[0] = data.read();

                }

                if(enable.read()==sc_logic_0)
                {
                        data_array[2] = sc_logic_0;
                        data_array[1] = sc_logic_0;
                        data_array[0] = sc_logic_0;
                }

                if((data_array[0] == sc_logic_1 || data_array[0] == sc_logic_0 ) & (data_array[1]
== sc_logic_0 || data_array[1] == sc_logic_1) & (data_array[2] == sc_logic_1 || data_array[2] ==
sc_logic_0)) valid_out.write(sc_logic_1);

                if((data_array[0] == sc_logic_1) & (data_array[1] == sc_logic_0) & (data_array[2]
== sc_logic_1)) seq_found.write(sc_logic_1);
                else seq_found.write(sc_logic_0);

                wait(10,SC_NS);

        }
}
```

Figure 10 shows the result of the simulation.

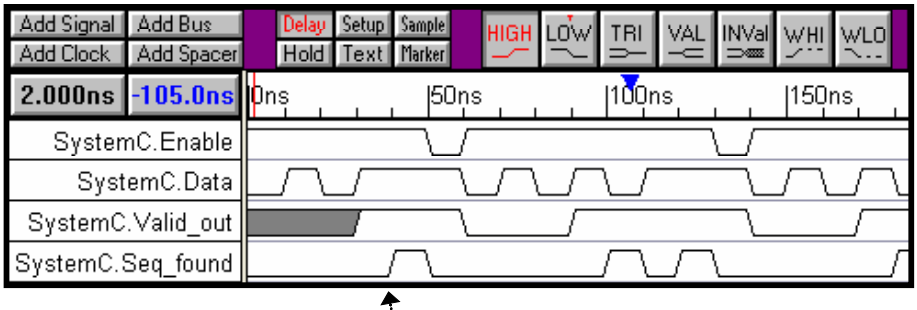


Figure 10 – Output waveform: sequence detector: behavioral level of abstraction

3.2.2 Sequence detector: implicit FSM level of abstraction

The use of the clock in the sequence_detector module is the main difference between the behavioral and the implicit FSM levels of abstraction. Figure 11 shows the class diagram of the sequence detector modeled at the implicit level, where the clock is synchronizing the different elements of the system. Also the implementation of the sequence_detector is performed by the control_path module and data_path module, which are instantiated by the sequence_detector module.

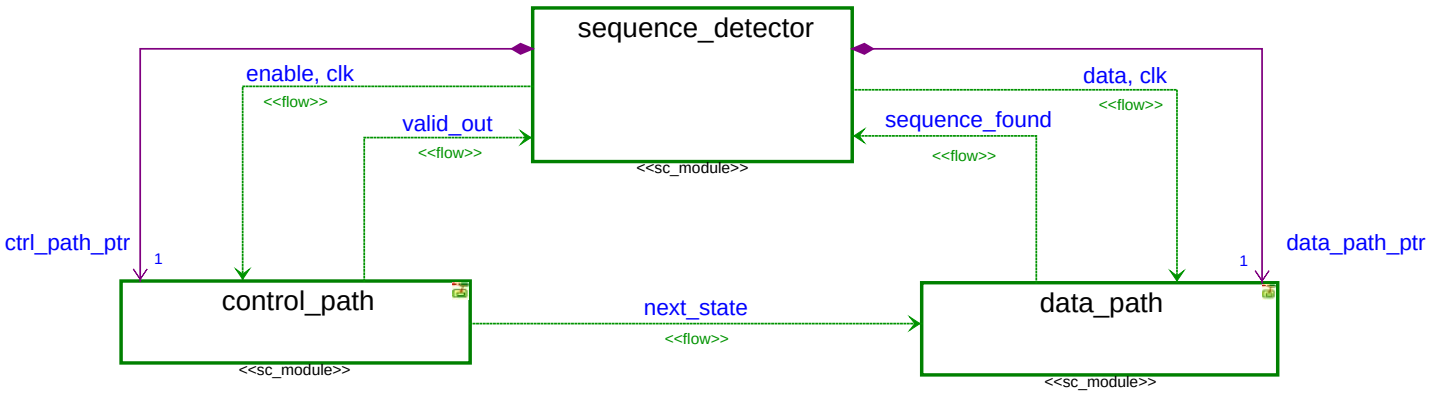


Figure 11 – Sequence detector at implicit FSM level: second level class diagram

The code below corresponds to the declaration of the sequence_detector module.

```
sequence_detector::sequence_detector(sc_module_name name) : sc_module(name) {
    initRelations();
    //#[ operation sequence_detector(sc_module_name)

    ctrl_path_ptr->clk ( clk );
    ctrl_path_ptr->enable ( enable );
    ctrl_path_ptr->next_state ( state );
    ctrl_path_ptr->valid_out ( valid_out );

    data_path_ptr->data ( data );
    data_path_ptr->next_state ( state );
    data_path_ptr->clk ( clk );
    data_path_ptr->sequence_found ( sequence_found );

    SC_METHOD(main_action);
    sensitive_pos<<clk;
    //#]
}
```

The code below corresponds to the implementation of the control_path module.

```
void control_path::main()
{
      switch (rootState_active)
      {
            case state_0:
                  if(isEnable()) rootState_active = state_1;
                  else if(isEnable()==false) rootState_active = state_0;
                  break;
             case state_1:
                   if(isEnable()==false) rootState_active = state_0;
                   else if(isEnable()) rootState_active = state_2;
                  break;
             case state_2:
                  if(isEnable()) rootState_active = state_3;
                  break;
             case state_3:
                   if(isEnable()==false)
                  {
                        valid_out.write(sc_logic_0);
                        rootState_active = state_0;
                  }
                  else valid_out.write(sc_logic_1);
                  break;
            default:
                  break;
      }
      assignNextState();
}
```

The code below corresponds to the implementation of the data_path module.

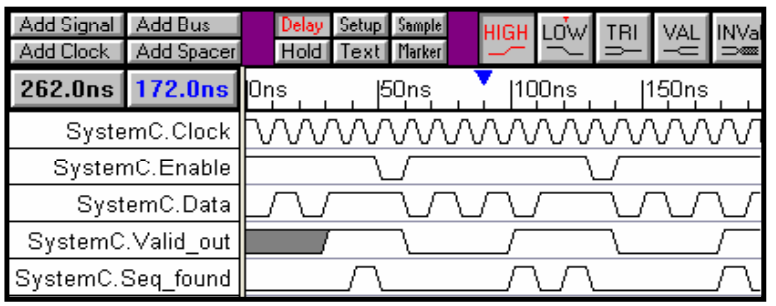
```
void data_path::main()
```

```
{

        switch (rootState_active)
        {
                case state_0:
                        if(isStateZero()) rootState_active = state_0;
                        else
        {
                                if(isStateZero()==false)
                                {
                                        assign();
                                        rootState_active = state_1;
                                        checkSequence();
          }
                        }
        break;

      case state_1:
                        if(isStateZero())
                        {
                                reset();
                                rootState_active = state_0;
                        }
                        else
                        {
                                if(isStateZero()==false)
                                {
                                        assign();
                                        rootState_active = state_1;
                                        checkSequence();
                                }
                        }
        break;
        default:
        break;
         }
}
```

Figure 12 shows the result of the simulation.



Output High after the 101 sequence is detected

Figure 12 – Output waveform: sequence detector: implicit FSM level of abstraction

3.2.3 Sequence detector: explicit FSM level of abstraction

Figure 13 show the class diagram of the sequence detector at the explicit level of abstraction. In this case, the control_path and the data_path modules are instantiating other modules that will be responsible for the implementation of the sequence detector.

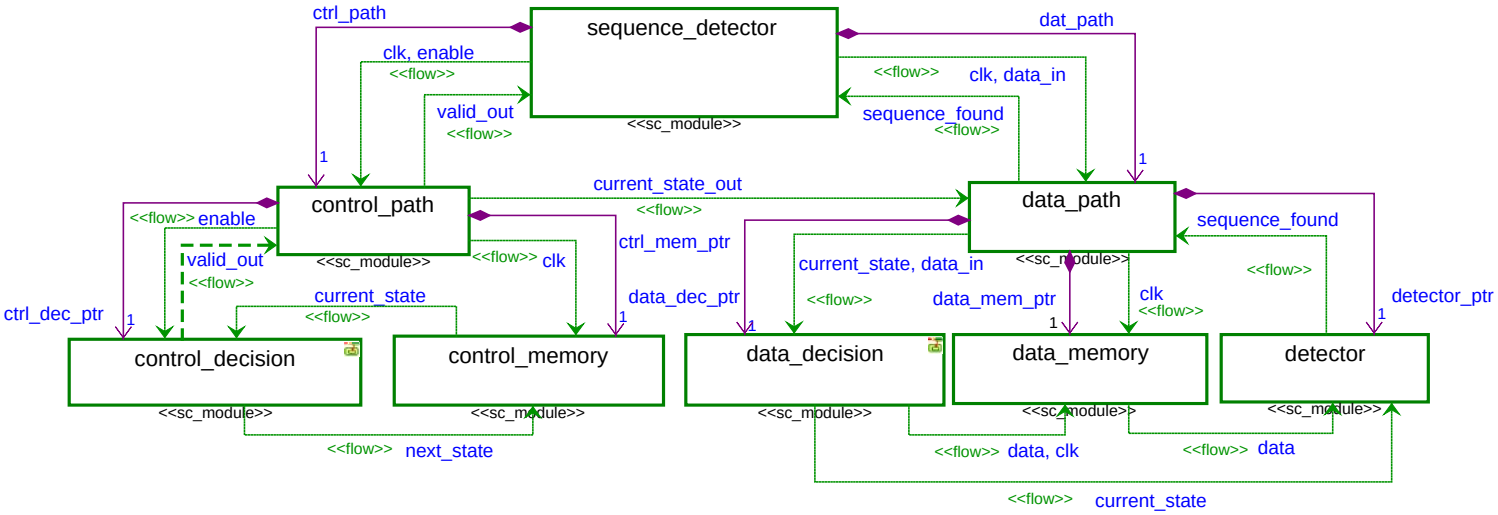


Figure 13 – Sequence detector at explicit FSM level: second level class diagram

The code below shows the declaration of the data_path module, which is instantiating and performing the port bindings of the data_desicion, data_memory, and detector modules.

```
SC_HAS_PROCESS ( data_path );

        data_path ( sc_module_name name ) : sc_module ( name )

        {
                data_dec_ptr = new data_decision ( "data_path_combinational_logic1" );
                data_dec_ptr->current_state ( current_state );
                data_dec_ptr->data_in ( data_in );
                data_dec_ptr->data_in_from_storage ( signal_data_out_storage );
                data_dec_ptr->data_out ( signal_data_out );

                data_mem_ptr = new data_memory ( "data_path_sl" );
                data_mem_ptr->data_in ( signal_data_out );
                data_mem_ptr->clk ( clk );
                data_mem_ptr->data_out ( signal_data_out_storage );

                detector_ptr = new detector ( "seq_det" );
```

```
        detector_ptr->data_in ( signal_data_out_storage );
        detector_ptr->current_state ( current_state );
        detector_ptr->sequence_found ( sequence_found );

    }
```

The code below shows the implementation of the control_decision module.

```
void control_decision::prc_output()
{
    if(current_state.read()==S4) valid_out.write(sc_logic_1);

    switch(current_state.read())
    {
        case S0:
            if(enable.read()==sc_logic_0) next_state.write(S0);
            else next_state.write(S1);
        break;
        case S1:
            if(enable.read()==sc_logic_0) next_state.write(S0);
            else next_state.write(S2);
        break;
        case S2:
            if(enable.read()==sc_logic_0) next_state.write(S0);
            else next_state.write(S3);
        break;
        case S3:
            if(enable.read()==sc_logic_0) next_state.write(S0);
            else next_state.write(S4);
        break;
        case S4:
            if(enable.read()==sc_logic_0) next_state.write(S0);
            else next_state.write(S4);
        break;
    }
```

```
}
```

The code below shows the implementation of the control_memory module.

```
void control_memory::prc_output()
{
        reg_current_state = next_state.read();

        current_state.write(reg_current_state);

}
```

The code below shows the implementation of the detector module.

```
void detector::prc_output ( )
{

        if(data_in.read() == "101") sequence_found.write(sc_logic_1);

        else sequence_found.write(sc_logic_0);

}
```

Figure 14 shows the result of the simulation.

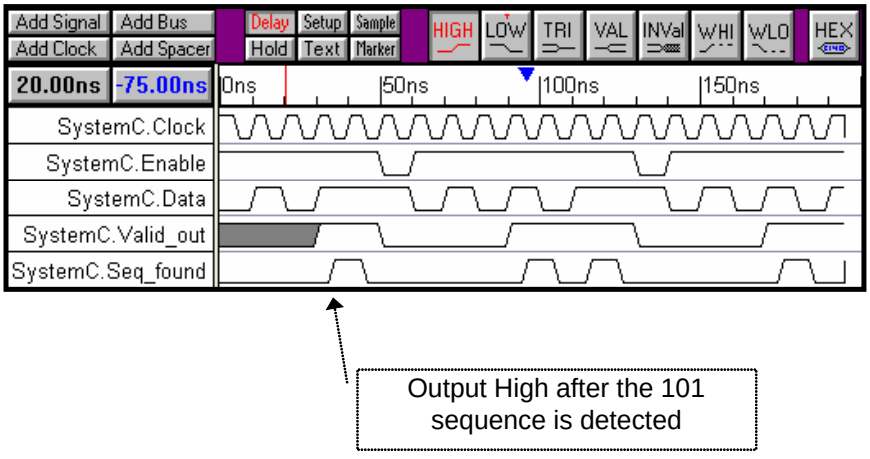


Figure 14 – Output waveform: sequence detector: explicit FSM level of abstraction

3.3 Parameterization Examples: Four-Bit Adder and Sequence Detector

3.3.1 Parameterization: Four-bit adder: behavioral, Boolean, gate

The goal in parameterization is to use….

Code snippet

four_bit_adder_main.cpp

```
int sc_main ( int argc,char* argv[] )
{

        clock_t start;
        clock_t finish;
        start = clock();                                                    // Starts timing

        sc_signal < sc_logic >  signal_cin;

        sc_signal < sc_logic >  signal_a0,signal_a1,signal_a2,signal_a3;

        sc_signal < sc_logic >  signal_b0,signal_b1,signal_b2,signal_b3;

                                                                            // Output signals
        sc_signal < sc_logic >  signal_sum0,signal_sum1,signal_sum2,signal_sum3;

        sc_signal < sc_logic >  signal_cout;

                                        // Instantiation and port binding of four_bit_adder_driver
module
        four_bit_adder_driver four_bit_adder_driver1 ( "GenerateWaveForms" );

        four_bit_adder_driver1.driver_a0 ( signal_a0 );
        four_bit_adder_driver1.driver_a1 ( signal_a1 );
        four_bit_adder_driver1.driver_a2 ( signal_a2 );
        four_bit_adder_driver1.driver_a3 ( signal_a3 );

        four_bit_adder_driver1.driver_b0 ( signal_b0 );
        four_bit_adder_driver1.driver_b1 ( signal_b1 );
        four_bit_adder_driver1.driver_b2 ( signal_b2 );
        four_bit_adder_driver1.driver_b3 ( signal_b3 );

        four_bit_adder_driver1.driver_cin ( signal_cin );

//      four_bit_adder four_bit_adder1 ( "four_bit_adder1" , BEHAVIORAL_LEVEL );
//      four_bit_adder four_bit_adder1 ( "four_bit_adder1" , BOOLEAN_LEVEL );
        four_bit_adder four_bit_adder1 ( "four_bit_adder1" , GATE_LEVEL );
```

```
   SC_HAS_PROCESS ( four_bit_adder );
```

```
        four_bit_adder ( sc_module_name name, level_of_abstraction level_abstraction ) :
sc_module ( name )
        {

                switch(level_abstraction)
                {

                        case BEHAVIORAL_LEVEL:
                                SC_METHOD ( prc_behavioral_level );

                                sensitive << fourBitinA1 << fourBitinB1 << fourBitcarry_in1;
                                sensitive << fourBitinA2 << fourBitinB2;
                                sensitive << fourBitinA3 << fourBitinB3;
                                sensitive << fourBitinA4 << fourBitinB4;

                        break;

                        case BOOLEAN_LEVEL:

                                full_adder_boolean_level *OneBitAdder1;
                                                        // Creating 4 i-bit full adders
                                full_adder_boolean_level *OneBitAdder2;
                                full_adder_boolean_level *OneBitAdder3;
                                full_adder_boolean_level *OneBitAdder4;

                                OneBitAdder1 = new full_adder_boolean_level ( "OneBitAdder1"
);

                                OneBitAdder1->a ( fourBitinA1 );            //Port bindings
                                OneBitAdder1->b ( fourBitinB1 );
                                OneBitAdder1->carry_in ( fourBitcarry_in1 );
                                OneBitAdder1->sum ( fourBitsum_out1 );
                                OneBitAdder1->carry_out ( oneBitcarry_out1 );

                                OneBitAdder2 = new full_adder_boolean_level ( "OneBitAdder2"
);

                                OneBitAdder2->a ( fourBitinA2 );            // Port bindings
                                OneBitAdder2->b ( fourBitinB2 );
                                OneBitAdder2->carry_in ( oneBitcarry_out1 );
                                OneBitAdder2->sum ( fourBitsum_out2 );
                                OneBitAdder2->carry_out ( oneBitcarry_out2 );

                                OneBitAdder3 = new full_adder_boolean_level ( "OneBitAdder3"
);

                                OneBitAdder3->a ( fourBitinA3 );                        // Port bindings
                                OneBitAdder3->b ( fourBitinB3 );
                                OneBitAdder3->carry_in ( oneBitcarry_out2 );
                                OneBitAdder3->sum ( fourBitsum_out3 );
                                OneBitAdder3->carry_out ( oneBitcarry_out3 );

                                OneBitAdder4 = new full_adder_boolean_level ( "OneBitAdder4"
);

                                OneBitAdder4->a ( fourBitinA4 );                        // Port bindings
```

```
                    OneBitAdder4->b ( fourBitinB4 );
                    OneBitAdder4->carry_in ( oneBitcarry_out3 );
                    OneBitAdder4->sum ( fourBitsum_out4 );
                    OneBitAdder4->carry_out ( fourBitcarry_out4 );

                    SC_METHOD ( prc_boolean_level );

                    sensitive << fourBitinA1 << fourBitinB1 << fourBitcarry_in1;
                    sensitive << fourBitinA2 << fourBitinB2;
                    sensitive << fourBitinA3 << fourBitinB3;
                    sensitive << fourBitinA4 << fourBitinB4;

            break;

            case GATE_LEVEL:

                    full_adder_gate_level *OneBitAdder01;   // Creating 4 i-bit full
adders
                    full_adder_gate_level *OneBitAdder02;
                    full_adder_gate_level *OneBitAdder03;
                    full_adder_gate_level *OneBitAdder04;

                    OneBitAdder01 = new full_adder_gate_level ( "OneBitAdder1" );

                    OneBitAdder01->a ( fourBitinA1 );          //Port bindings
                    OneBitAdder01->b ( fourBitinB1 );
                    OneBitAdder01->carry_in ( fourBitcarry_in1 );
                    OneBitAdder01->sum ( fourBitsum_out1 );
                    OneBitAdder01->carry_out ( oneBitcarry_out1 );

                    OneBitAdder02 = new full_adder_gate_level ( "OneBitAdder2" );

                    OneBitAdder02->a ( fourBitinA2 );          // Port bindings
                    OneBitAdder02->b ( fourBitinB2 );
                    OneBitAdder02->carry_in ( oneBitcarry_out1 );
                    OneBitAdder02->sum ( fourBitsum_out2 );
                    OneBitAdder02->carry_out ( oneBitcarry_out2 );

                    OneBitAdder03 = new full_adder_gate_level ( "OneBitAdder3" );

                    OneBitAdder03->a ( fourBitinA3 );          // Port bindings
                    OneBitAdder03->b ( fourBitinB3 );
                    OneBitAdder03->carry_in ( oneBitcarry_out2 );
                    OneBitAdder03->sum ( fourBitsum_out3 );
                    OneBitAdder03->carry_out ( oneBitcarry_out3 );

                    OneBitAdder04 = new full_adder_gate_level ( "OneBitAdder4" );

                    OneBitAdder04->a ( fourBitinA4 );          // Port bindings
                    OneBitAdder04->b ( fourBitinB4 );
                    OneBitAdder04->carry_in ( oneBitcarry_out3 );
                    OneBitAdder04->sum ( fourBitsum_out4 );
                    OneBitAdder04->carry_out ( fourBitcarry_out4 );

                    SC_METHOD ( prc_gate_level );
```

```
                                    sensitive << fourBitinA1 << fourBitinB1 << fourBitcarry_in1;
                                    sensitive << fourBitinA2 << fourBitinB2;
                                    sensitive << fourBitinA3 << fourBitinB3;
                                    sensitive << fourBitinA4 << fourBitinB4;

                                break;
                        }
                // End of CASE

                }
                // End of four_bit_adder() declaration

};
```

3.3.2 Parameterization: Sequence detector: behavioral, implicit FSM, explicit FSM

```
int sc_main ( int argc,char* argv[] )
{
        clock_t start;
        clock_t finish;
        start = clock();    //starts timing

        cout<<endl<<endl<<"Sequence detector"<<endl<<endl;

        sc_signal<sc_logic> signal_enable, signal_data, signal_valid_out, signal_seq_found;

        sc_clock clk("clk",10,SC_NS,FALSE);

        sequence_detector_driver sequence_detector_driver1 ( "SeqDetDriver" );
        sequence_detector_driver1.driver_enable ( signal_enable );
        sequence_detector_driver1.driver_data ( signal_data );
        sequence_detector_driver1.clk ( clk );

//      sequence_detector sequence_detector1 ( "sequence_detector1" , BEHAVIORAL_LEVEL
);
        sequence_detector sequence_detector1 ( "sequence_detector1" , IMPLICIT_FSM );
//      sequence_detector sequence_detector1 ( "sequence_detector1" , EXPLICIT_FSM );

        sequence_detector1.enable ( signal_enable );
        sequence_detector1.data ( signal_data );
        sequence_detector1.clk ( clk );                                         //For
implicit/explicit only
        sequence_detector1.valid_out ( signal_valid_out );
        sequence_detector1.seq_found ( signal_seq_found );

        sequence_detector_monitor sequence_detector_monitor1( "MonitorWaveForms" );
        sequence_detector_monitor1.monitor_enable ( signal_enable );
        sequence_detector_monitor1.monitor_data ( signal_data );
        sequence_detector_monitor1.monitor_clk ( clk );                 // For implicit/explicit
only
        sequence_detector_monitor1.monitor_valid_out ( signal_valid_out );
        sequence_detector_monitor1.monitor_seq_found ( signal_seq_found );

        sequence_detector_monitor1.trace_signals ( );    // Tracing signals
```

```
	sc_start ( 50000, SC_NS );

	finish = clock();  //stop timing

	clock_t computation_time = finish-start;

	cout<<endl<<endl<<"Simulation time :
"<<((long)computation_time)/((double)CLOCKS_PER_SEC)<<endl<<endl;

	return ( 0 );

}
	// End of sc_main
```

sequence_detector.h

```
	SC_HAS_PROCESS(sequence_detector);

	sequence_detector ( sc_module_name name, level_of_abstraction level_abstraction ) :
sc_module ( name )
	{

		switch(level_abstraction)
		{

			case BEHAVIORAL_LEVEL:

				SC_THREAD(prc_seq_det_behavioral);

			break;

			case IMPLICIT_FSM:

				initRelations();

				ctrl_path_implicit_ptr->clk ( clk );
				ctrl_path_implicit_ptr->enable ( enable );
				ctrl_path_implicit_ptr->next_state ( signal_current_state );
				ctrl_path_implicit_ptr->valid_out ( valid_out );

				data_path_implicit_ptr->data ( data );
				data_path_implicit_ptr->next_state (signal_current_state );
				data_path_implicit_ptr->clk ( clk );
				data_path_implicit_ptr->sequence_found ( sequence_found );

			break;

			case EXPLICIT_FSM:

				initRelations();

				control_path_explicit_ptr->enable ( enable );
				control_path_explicit_ptr->clk ( clk );
				control_path_explicit_ptr->valid_out ( valid_out );
```

```
                                    control_path_explicit_ptr->current_state_out (
signal_current_state );

                                    data_path_explicit_ptr->current_state ( signal_current_state );
                                    data_path_explicit_ptr->data_in ( data );
                                    data_path_explicit_ptr->clk ( clk );
                                    data_path_explicit_ptr->sequence_found ( sequence_found );

                            break;
                    }


            }
};
```

3.4 Robot controller: A Concurrent Design Example

Mention that it is from Grotker's book. Copy some lines from there. I will modify them later tonight.

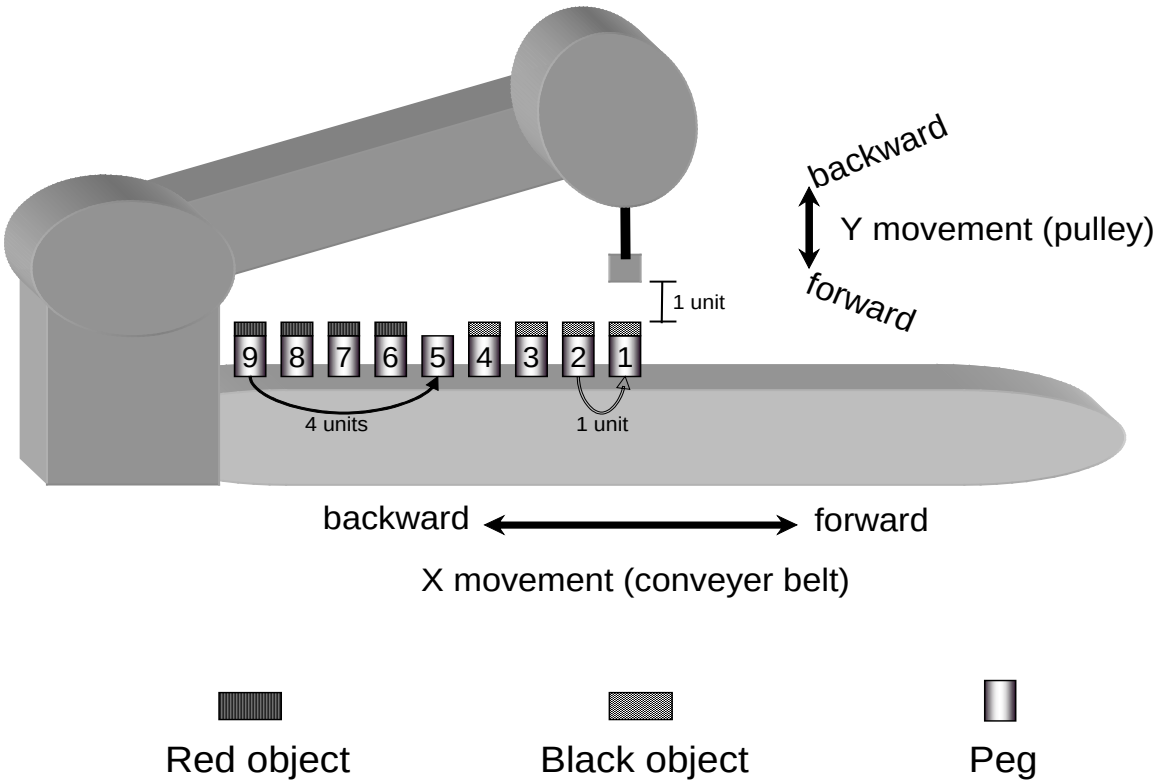
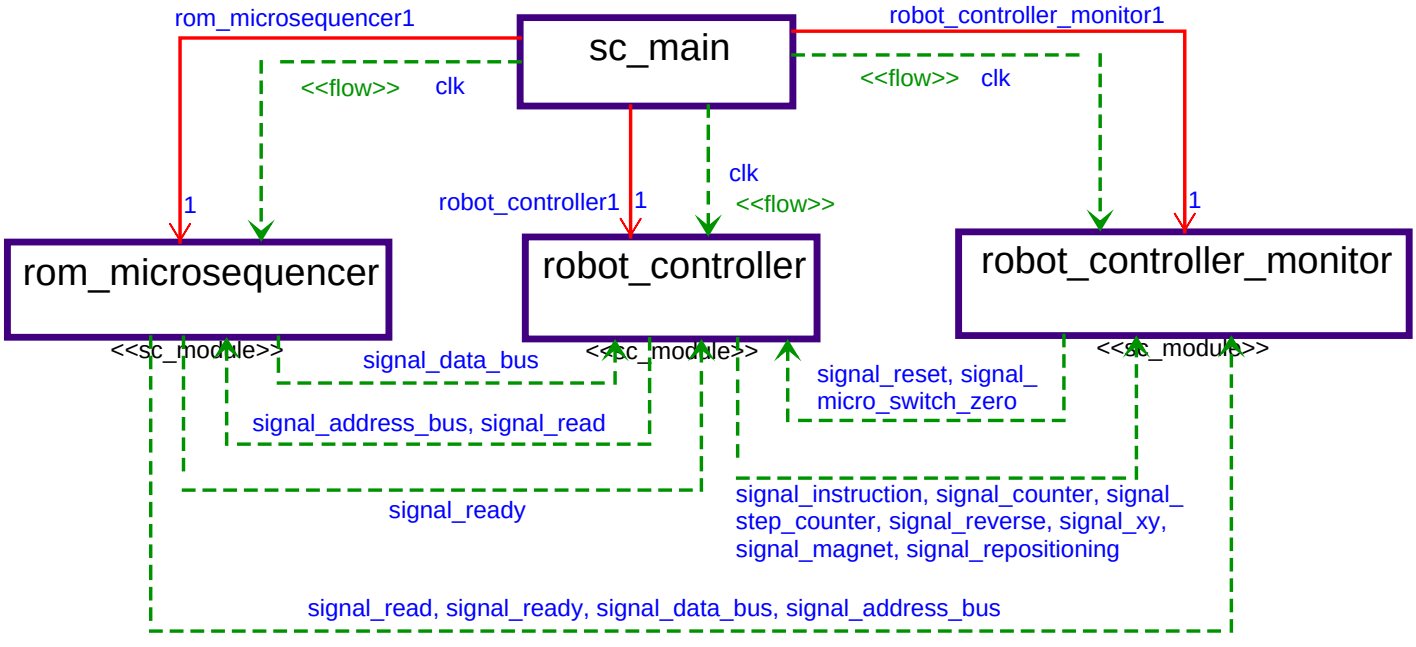


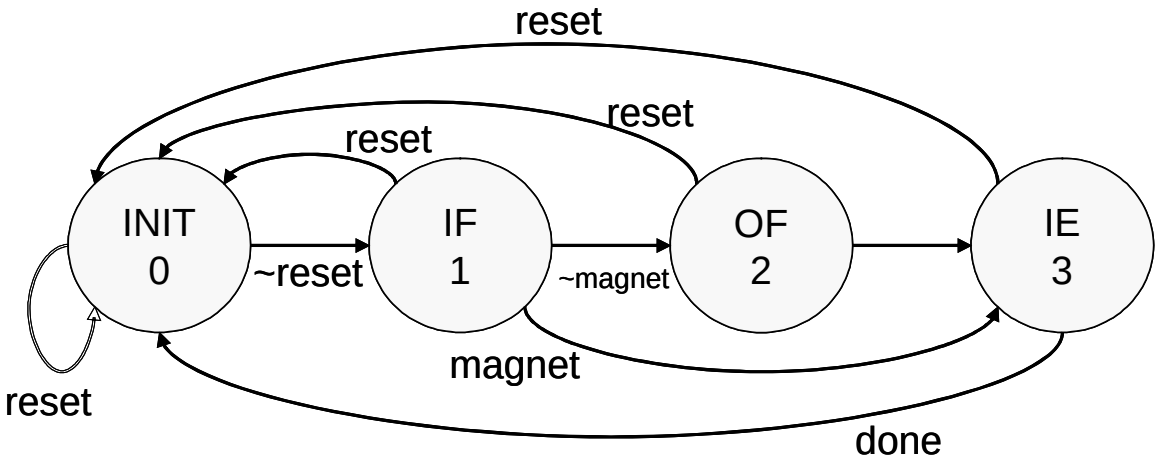
Figure 15 – Robot controller

Class diagram



State Machine

Figure 16 – Robot controller: top level class diagram



IF = instruction fetch

OF = operand fetch

IE = instruction execute

Figure 17 – Robot controller: state machine

Code snippet

rom_microsequencer.h

```
SC_MODULE(rom_microsequencer)
{
        public:

                sc_in < bool > clk;
                sc_in < sc_uint < ADDRESS_SIZE > > address_bus;
                sc_in < bool > read;
                sc_inout < bool > ready;
                sc_out < sc_uint < BUS_SIZE > > data_bus;

                sc_uint < WIDTH > mem[MEMORY_SIZE];
                sc_uint < ADDRESS_SIZE > address;

                void store_microsequence();
                void main_action();

        SC_HAS_PROCESS(rom_microsequencer);
        rom_microsequencer(sc_module_name name):sc_module(name)
        {
```

```
        SC_THREAD(store_microsequence);

        SC_METHOD(main_action);
        sensitive_neg<<clk;

    }

};
```

rom_microsequencer.cpp

```
void rom_microsequencer::main_action()
{
    if(ready.read()==false) ready.write(true);

    if(read.read())
    {
        address = address_bus.read();
        data_bus.write(mem[address]);
    }
}

void rom_microsequencer::store_microsequence()
{
    data_bus.write(15);

    mem[0]=0x00;   // Step X forward
    mem[1]=0x03;   // 3 steps
    mem[2]=0x04;   // Step Y forward
    mem[3]=0x01;   // 1 step
    mem[4]=0x02;   // Turn ON electromagnet
    mem[5]=0x0C;   // Step Y backward
    mem[6]=0x01;   // 1 step
    mem[7]=0x00;   // Step X forward
    mem[8]=0x01;   // 1 step
    mem[9]=0x04;   // Step Y forward
    mem[10]=0x01; // 1 step
    mem[11]=0x03; // Turn OFF electromagnet
    mem[12]=0x0C;// Step Y backward
    mem[13]=0x01; // 1 step

    ….
    ….
    ….
    ….
```

robot_controller.h

```
SC_HAS_PROCESS ( robot_controller );
    robot_controller ( sc_module_name name ) : sc_module ( name )
    {
        SC_THREAD (init);

        SC_METHOD ( counter_proc );
```

```
                sensitive_pos<< clk;

                SC_METHOD ( control_fsm_state );
                sensitive_pos<< clk;

                SC_METHOD ( control_fsm );
                sensitive << reset << repositioning << magnet << done << micro_switch_zero;
                sensitive << current_state;
                sensitive << data_bus;
                sensitive_pos << ready;

        }
```

robot_controller.cpp

```
void robot_controller::init ( )
{
        current_state = 0;
}

void robot_controller::counter_proc ( )
{
        if ( count.read() )
        {
                step_counter.write( counter[0] );
                counter = counter - 1;
                if(counter == 0)
                {
                        done.write(true);

                }
        }

        if(counter == 0)
        {
                count.write(false);
        }
}

void robot_controller::control_fsm_state ( )
{
        current_state.write(next_state.read());
}

void robot_controller::control_fsm ( )
{
        count.write(false);

        out_current_state.write(current_state);

        if( reset.read() == false ) next_state = INIT;

        switch(current_state.read())
                {
```

```
                        case INIT:

                                step_counter.write(false);
                                done.write(false);
                                magnet.write(false);
                                repositioning.write(false);
                                xy.write(false);
                                reverse.write(false);
                                read.write(false);

                                next_state = IF;

                        break;

                        case IF:

                                if(read.read() == false)
                                {

                                        memory_address_register = program_counter;
                                        address_bus.write(memory_address_register);
                                        program_counter = program_counter + 1;
                                        read.write(true);
                                        ready.write(false);

                                }

                                else
                                {

                                        memory_data_register = data_bus.read();
                                        instruction = memory_data_register;
                                        out_instruction.write(memory_data_register);

                                        if(instruction==8 || (instruction==0) || instruction==4 ||
instruction == 12)
                                        {
                                                next_state = OF;
                                                read.write(false);

                                        }

                                        else
                                        {
                                                read.write(false);
                                                next_state = IE;
                                                if(         instruction == 1 || instruction == 13)
counter = 512;
                                        }
                                }


                        break;

                        case OF:
```

```
                    if(read.read() == false)
                    {
                            memory_address_register = program_counter;
                            address_bus.write(memory_address_register);
                            program_counter = program_counter + 1;

                            read.write(true);
                            ready.write(false);


                    }

                    else
                    {
                            memory_data_register = data_bus.read();
                            memory_data_register = memory_data_register * 3;
                            counter = memory_data_register;
                            out_counter.write(memory_data_register);
                            next_state = IE;
                    }


            break;

            case IE:

                    repositioning.write((sc_bit)instruction[0]);
                    magnet.write((sc_bit)instruction[1]);
                    xy.write((sc_bit)instruction[2]);
                    reverse.write( (sc_bit)instruction[3]);

                    if( magnet.read() || repositioning.read() )
                    {
                            next_state = INIT;
                    }
                    else count.write(true);

                    if ( done.read() )
                    {
                            next_state = INIT;
                            count.write(false);

                    }

                    else count.write(true);

                    if(instruction == 1 || instruction == 13)
                    {
                            if(micro_switch_zero.read())
                            {
                                    counter = 0;
                            }
                    }

            break;
```

```
                }
}
```

Results: output screen

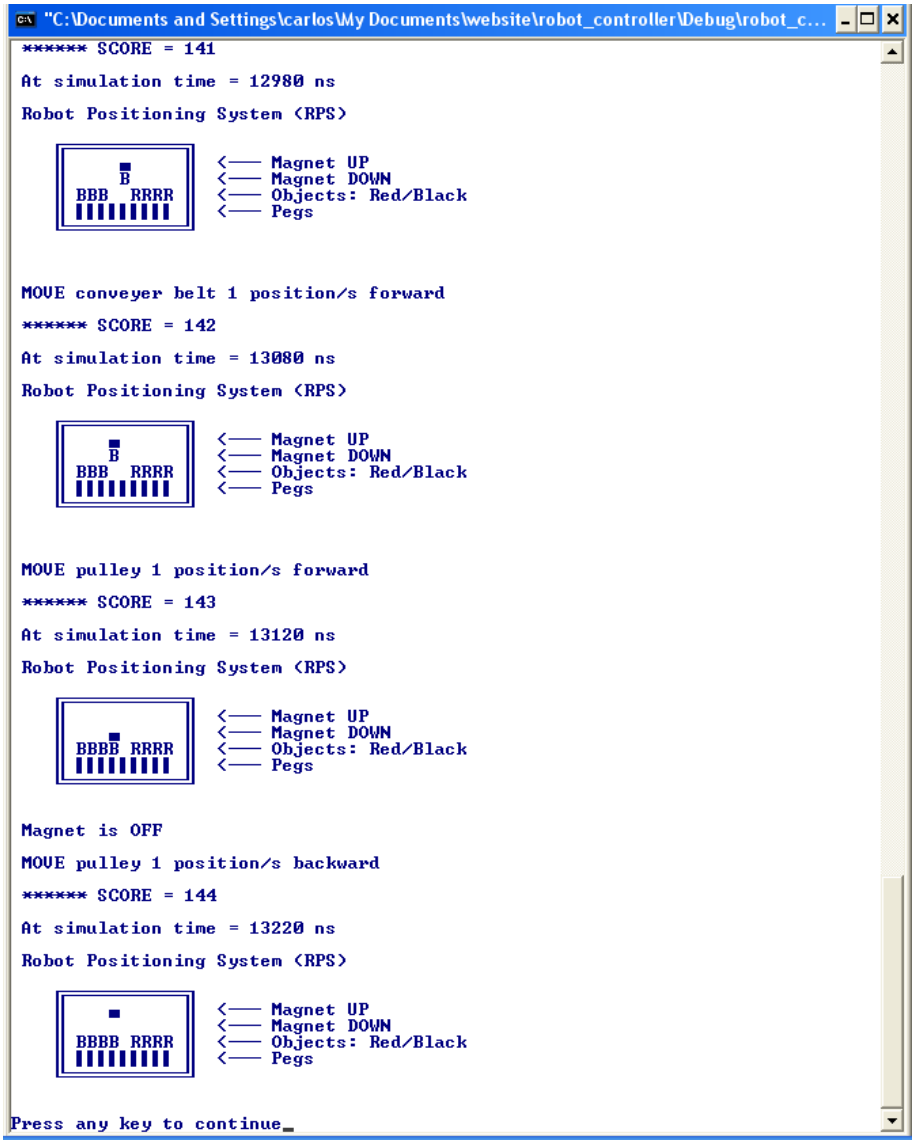


Figure 18– Robot controller: screen output

Results: output waveform

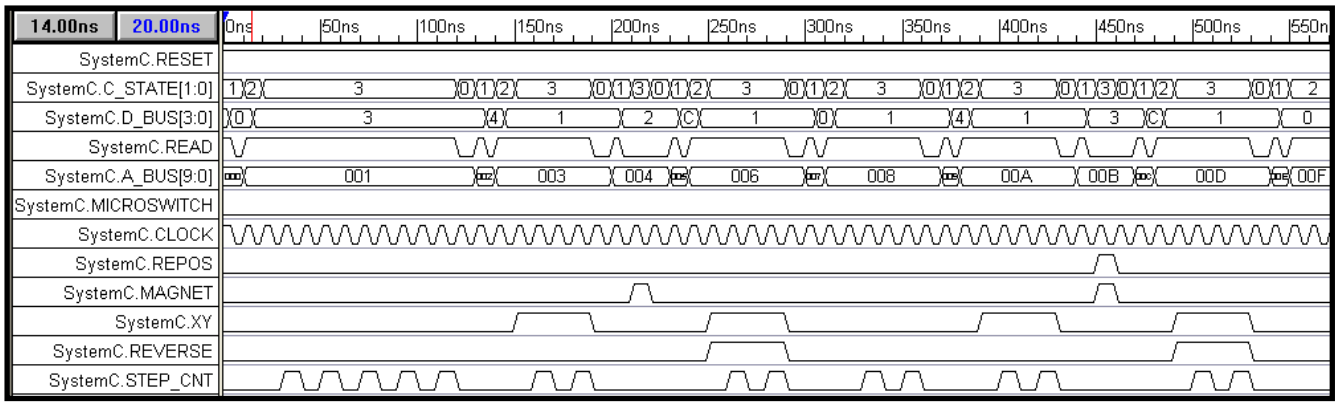


Figure 19– Robot controller: output waveform

3.5 Simple CPU: Two Levels of abstraction

3.5.1 Simple CPU: Behavioral level of abstraction

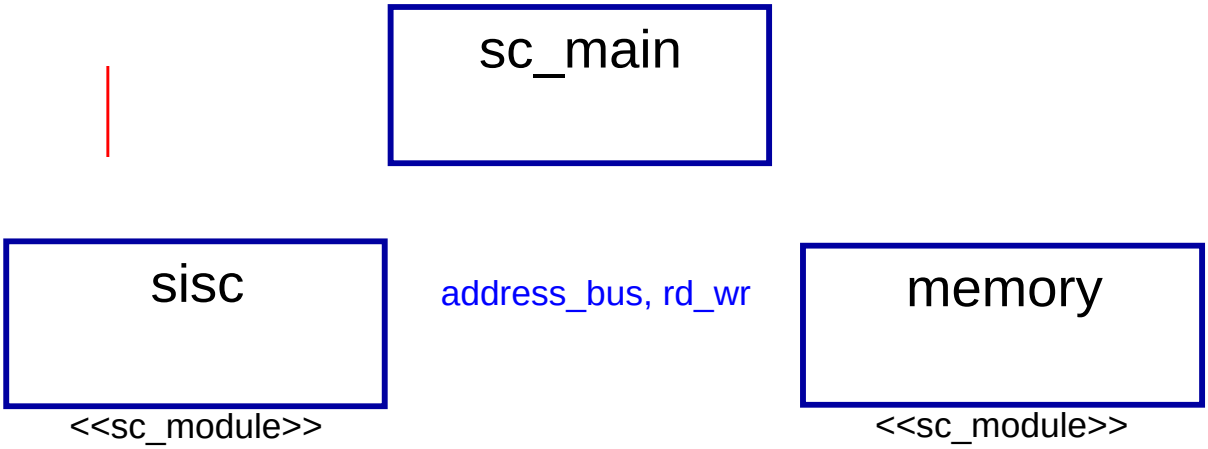


Figure 20– Simple CPU: top level class diagram

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

        SC_HAS_PROCESS(memory);
        memory(sc_module_name name):sc_module(name)
```

```
{

//          initTestInstructions();

            init();

            SC_METHOD(main_action);
            sensitive_neg<<clk;
}

void main_action()
{
            adrs = address_bus;
            //if (rd_wr.read() == sc_logic ('1'))   //READ operation
            if (rd_wr== sc_logic ('1'))
            {
                        data_bus = mem[adrs];
                        //cout<<"Now READING IN MEMORY"<<endl;
            }
            else //WRITE Operation
            {
                        if(rd_wr == sc_logic ('0'))
                        {
                                    data = data_bus;
                                    mem[adrs] = data;
                                    cout<<"NOW WRITING TO MEMORY"<<endl;
                        }
            }
}

void init()
{

            mem[0]=0x69; //STORE #8,$0x50
            mem[1]=0x08;
            mem[2]=0x50;
            mem[3]=0x69; //STORE #10,$0x51;
            mem[4]=0x0A;
            mem[5]=0x51;
            mem[6]=0x69; //STORE #13,$0x52;
            mem[7]=0x0D;
            mem[8]=0x52;
            mem[9]=0x54; //LOAD $0x50, R0;
            mem[10]=0x50;
            mem[11]=0x00;
            mem[12]=0x54; //LOAD $0x51 , R1
            mem[13]=0x51;
            mem[14]=0x01;
            mem[15]=0x54; //LOAD $0x52 , R2;
            mem[16]=0x52;
            mem[17]=0x02;
            mem[18]=0x70; //CMP R1,R0;
            ….
            ….
            ….
```

sisc.h

```
#define AND             0       //0000
#define OR              1       //0001
#define ADD             2       //0010
#define SUB             3       //0011
#define MOV             4       //0100
#define LOAD    5       //0101
#define STORE 6         //0110
#define CMP             7       //0111
#define JMP             8       //1000
#define JGT             9       //1001
#define JNE             10      //1010
#define NOP             14      //1110
#define HALT    15      //1111

#define R0                              0
#define R1                              1
#define R2                              2
#define R3                              3

#define Register_Mode  0
#define Direct_Mode             1
#define Immediate_Mode          2


SC_MODULE(sisc)
{
        sc_out<sc_uint<ADDRS_SIZE> > address_bus;
        sc_out<sc_logic> rd_wr;
        sc_inout<sc_uint<WIDTH> > data_bus;

        SC_HAS_PROCESS(sisc);

        sisc(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);
                }
        }

        ……
```

```
……
……
……

        switch (opcode)
        {

        case ADD :

                                cout<<"------ ADD ------"<<endl<<endl;

                                cout<<"Source = "<<source<<endl;
                                cout<<"Destination = "<<destination<<endl;

                                update_SR();

                                destination = destination + source;

                                cout<<"Result  -->  "<<destination<<endl;

                                write_result();

                                break;

……
……
……
……
```

Results: output screen

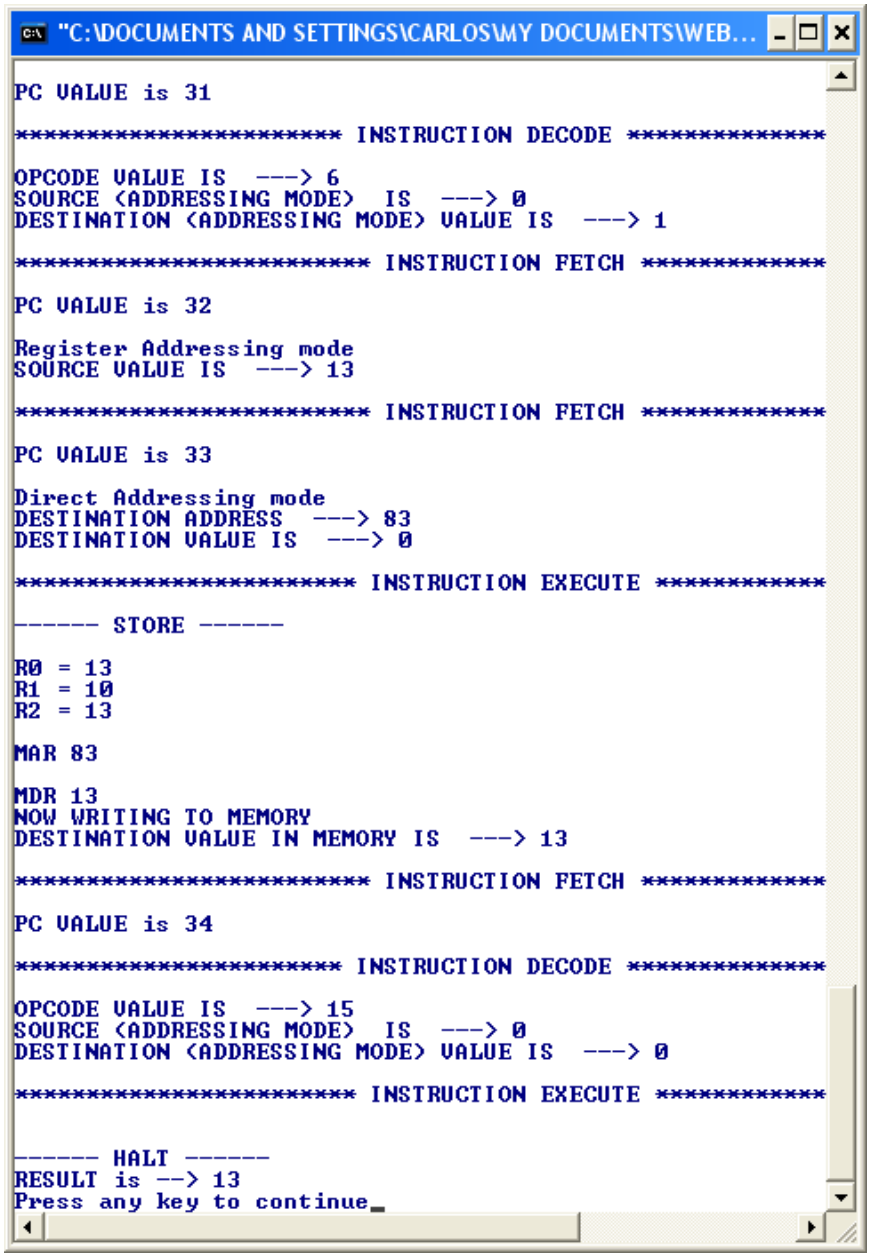


Figure 21 – Simple CPU: behavioral level: Screen output

Results: output waveform

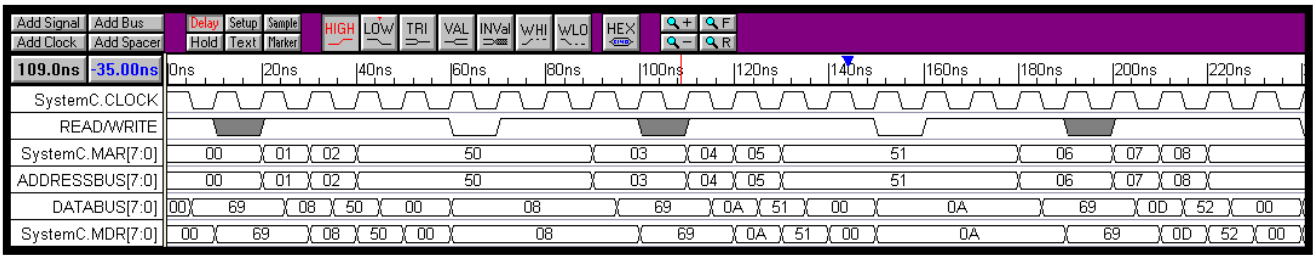


Figure 22 – Simple CPU at behavioral level of abstraction: output waveform

3.5.2 Simple CPU: cycle accurate level of abstraction

Class diagram

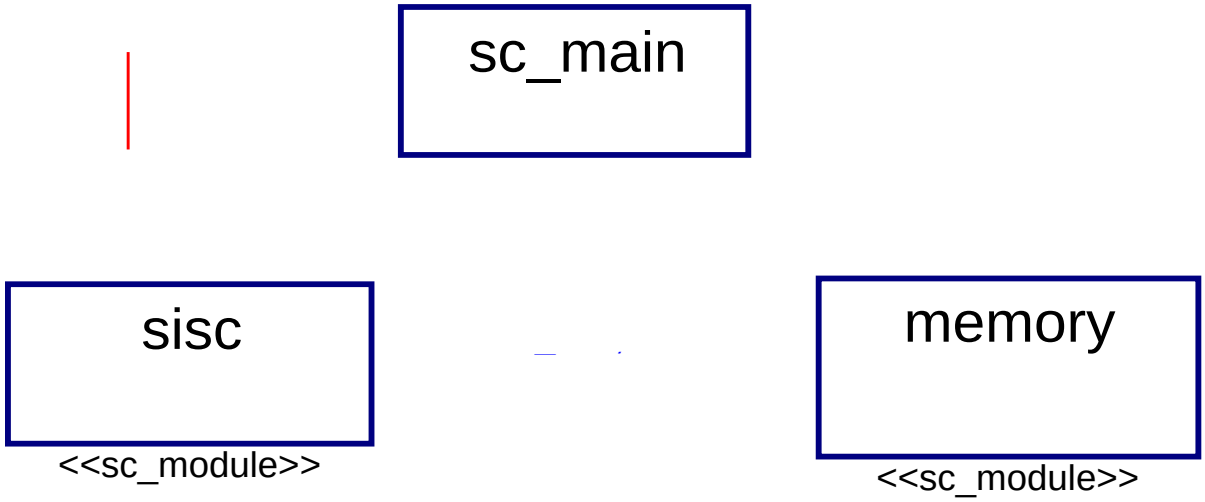


Figure 23 – Simple CPU at cycle accurate level of abstraction: class diagram

Code snippet

memory.h

```
SC_MODULE(memory)
{
        sc_in<sc_uint<ADDRS_SIZE> > address_bus;
        sc_in<sc_logic> rd_wr;
        sc_inout<sc_uint<WIDTH> > data_bus;
        sc_inout<bool> dtack;
```

```
        sc_in<bool> clk;

        SC_HAS_PROCESS(memory);
        memory(sc_module_name name):sc_module(name)
        {
                init();

                //initTestInstructions();

                SC_METHOD(main_action);
                sensitive_neg<<clk;
        }

        public:

                void main_action();

                void init();

                void initTestInstructions();


        private:
                sc_uint<WIDTH> mem[MEM_SIZE], data;
                sc_uint<ADDRS_SIZE> adrs;

}; //END of MEMORY Module
```

memory.h

```
void memory::main_action()
{
        adrs = address_bus;
        //READ operation
        if (rd_wr== sc_logic ('1') )
        {
                data_bus = mem[adrs];
                dtack=true;
        }
        else //WRITE Operation
        {
                if(rd_wr == sc_logic ('0') )
                {
                        data = data_bus;
                        mem[adrs] = data;
                        cout<<"NOW WRITING TO MEMORY"<<endl;

                }
        }
}

/******************************** init() *********************************/

void memory::init()
{
```

```
        mem[0]=0x69; //STORE #8,$0x50
        mem[1]=0x08;
        mem[2]=0x50;
        mem[3]=0x69; //STORE #10,$0x51;
        mem[4]=0x0A;
        mem[5]=0x51;
        mem[6]=0x69; //STORE #13,$0x52;
        mem[7]=0x0D;
        mem[8]=0x52;
        mem[9]=0x54; //LOAD $0x50, R0;
        mem[10]=0x50;
        mem[11]=0x00;
        mem[12]=0x54; //LOAD $0x51 , R1
        …..
        …..
        …..

}
```

sisc.h

```
#define AND             0       //0000
#define OR              1       //0001
#define ADD             2       //0010
#define SUB             3       //0011
#define MOV             4       //0100
#define LOAD    5       //0101
#define STORE 6         //0110
#define CMP             7       //0111
#define JMP             8       //1000
#define JGT             9       //1001
#define JNE             10      //1010
#define NOP             14      //1110
#define HALT    15      //1111

#define R0                              0
#define R1                              1
#define R2                              2
#define R3                              3

#define INIT                    0
#define IF                              1
#define OF11                    2
#define OF12                    3
#define OF21                    4
#define OF22                    5
#define IE                              6
#define OS                              7


#define Register_Mode  0
#define Direct_Mode             1
#define Immediate_Mode          2
```

```
SC_MODULE(sisc)
{
        sc_in<bool> clk;

        sc_out<sc_uint<ADDRS_SIZE> > address_bus;
        sc_out<sc_logic> rd_wr;

        sc_inout<sc_uint<WIDTH> > data_bus;
        sc_inout<bool> dtack;

        sc_signal<sc_uint<3> > current_state, next_state;

        SC_HAS_PROCESS(sisc);
        sisc(sc_module_name name):sc_module(name)
        {
                SC_METHOD(initialize);

                SC_METHOD(fsm_next_state);
                sensitive_pos << clk;

                SC_METHOD(fsm_states);
                sensitive<<current_state;
                sensitive_pos<<dtack;
        }

……
……
```

sisc.cpp

```
/******************************** initialize() *********************************/

void sisc::initialize()
{
        PC = 0;
        SR = 0;
        next_state = IF;
        instr_flag=true;
        oper_flag=true;
        oper12_flag=true;
        mem_flag=true;
}

/******************************** fsm_next_state() *********************************/


void sisc::fsm_next_state()
{
        current_state.write( next_state.read() );
}

/******************************** fsm_states() *********************************/
```

```
void sisc::fsm_states()
{
        switch ( current_state.read() )
        {
                case INIT: break;

                case IF:

                        if ( instr_flag ) instr_fetch();

                        else
                        {
                                instr_decode();
                                if ( opcode == NOP ) next_state = IE;

                                else
                                {
                                        next_state = OF11;
                                        oper_flag=true;
                                }
                        }
                        break;

                …..
                ……

                case IE:

                        instr_exec();

                        if ( opcode == NOP ) next_state = IF;
                        else next_state = OS;

                        break;

                case OS:

                        if ( opcode.range(3,2).or_reduce()==0 )
                        {
                                write_result();
                        }
                        else if ( opcode==LOAD )
                        {
                                GPR[dest_addr] = data_bus.read();
                                cout<<"R0 = "<<GPR[R0]<<endl;
                                cout<<"R1 = "<<GPR[R1]<<endl;
                                cout<<"R2 = "<<GPR[R2]<<endl;
                        }
                        else if ( opcode==HALT )
                        {
                                cout<<"Result is:"<<data_bus.read()<<endl;
                                exit(0);
                        }
                        next_state = IF;

                        break;
```

```
                default:
                        cout<<"Incorrect state. Exiting";
                        exit(0);
        }
}
```

Results: output waveform

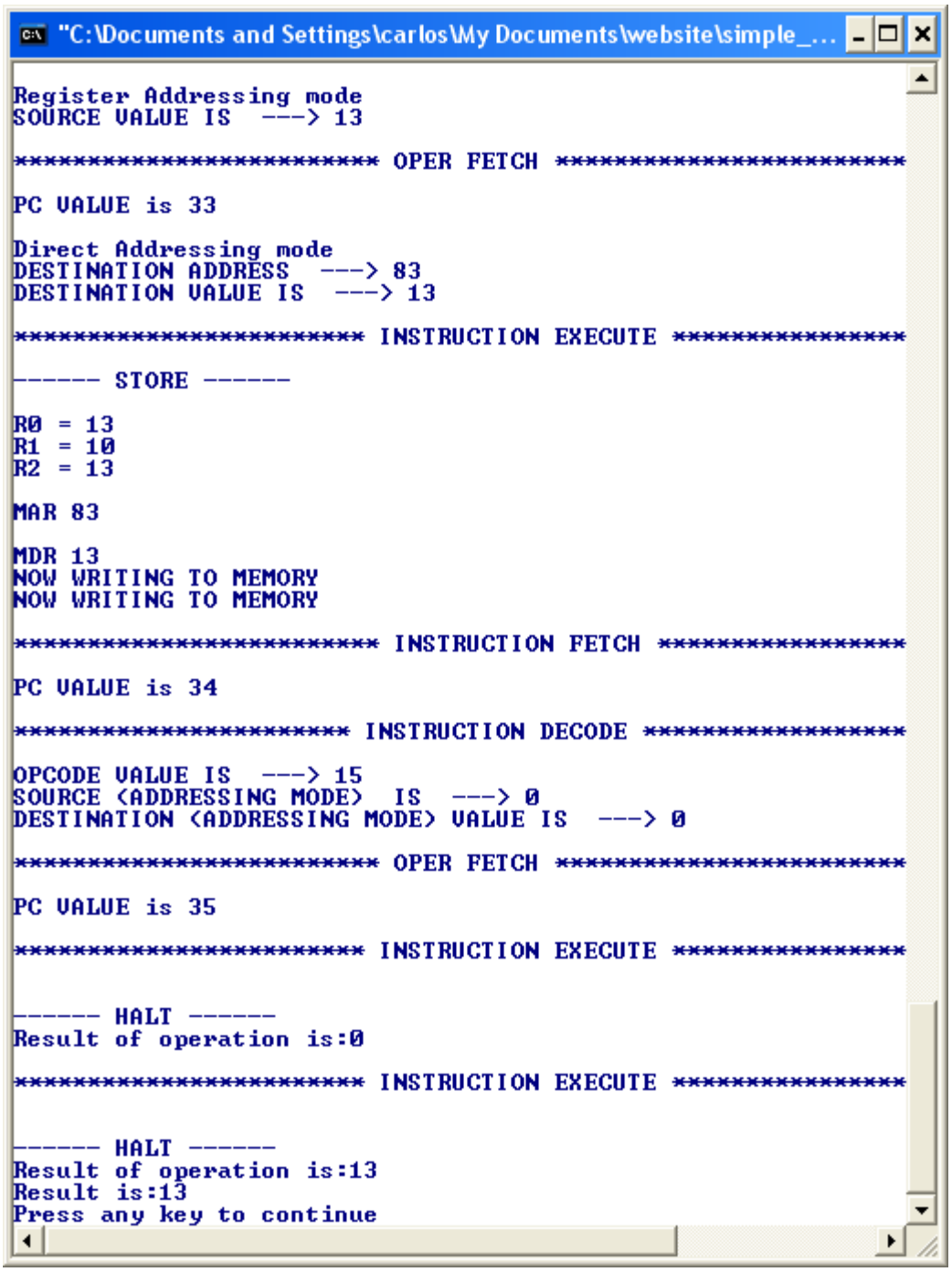


Figure 24 – Simple CPU at cycle accurate level of abstraction: screen output

Results: output waveform

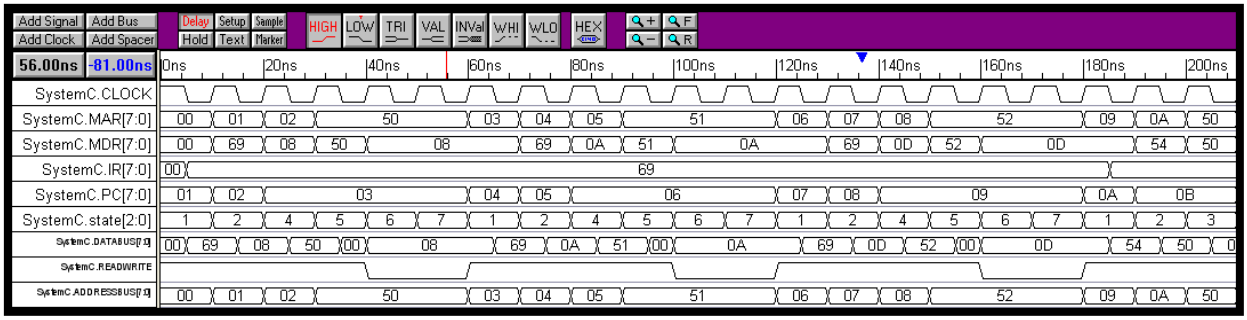


Figure 25 – Simple CPU at cycle accurate level of abstraction: output waveform

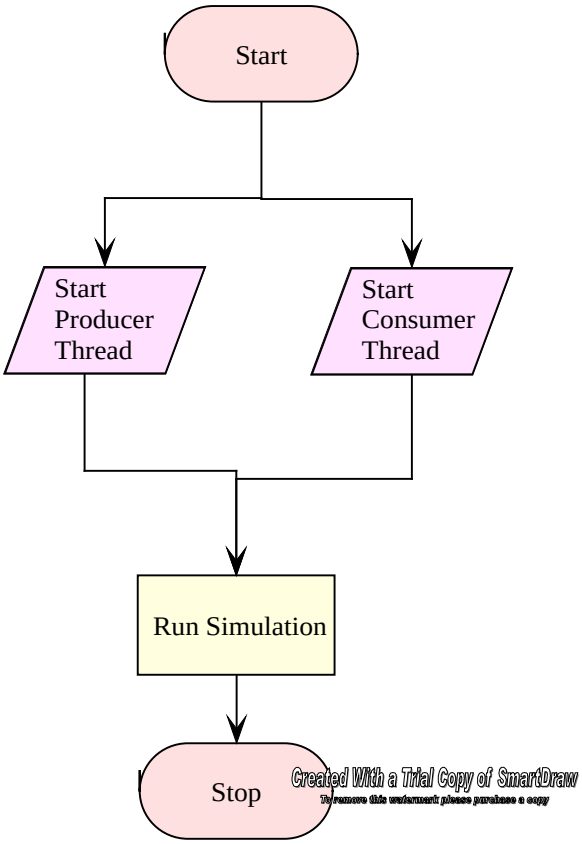
3.6 Producer / consumer example

<u>Producer/Consumer Example using Methods</u>

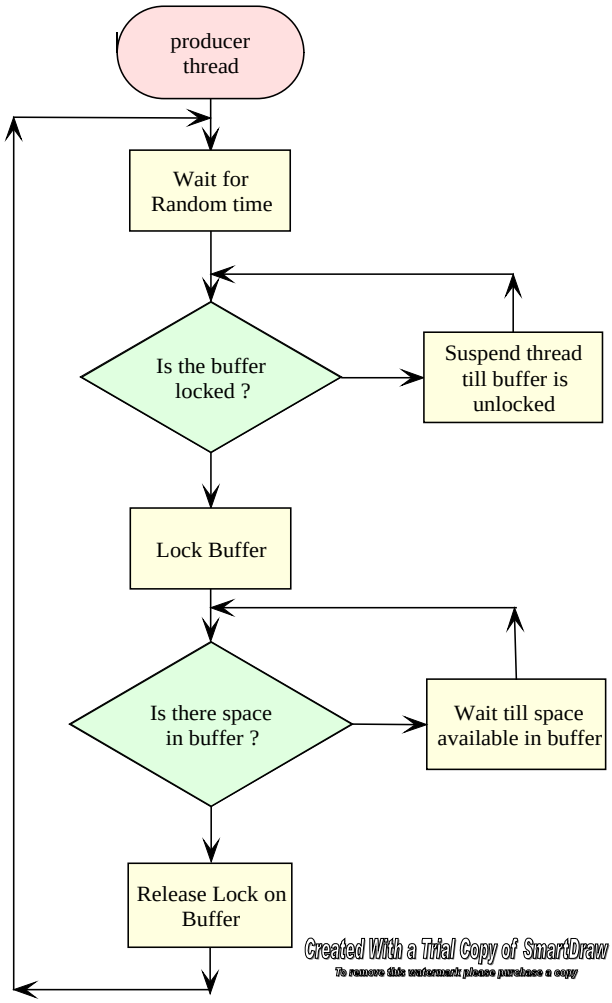
Carlos, Include a short discussion on the producer-consumer example, as covered in Grotker's book.

(A) Flowchart for Main method

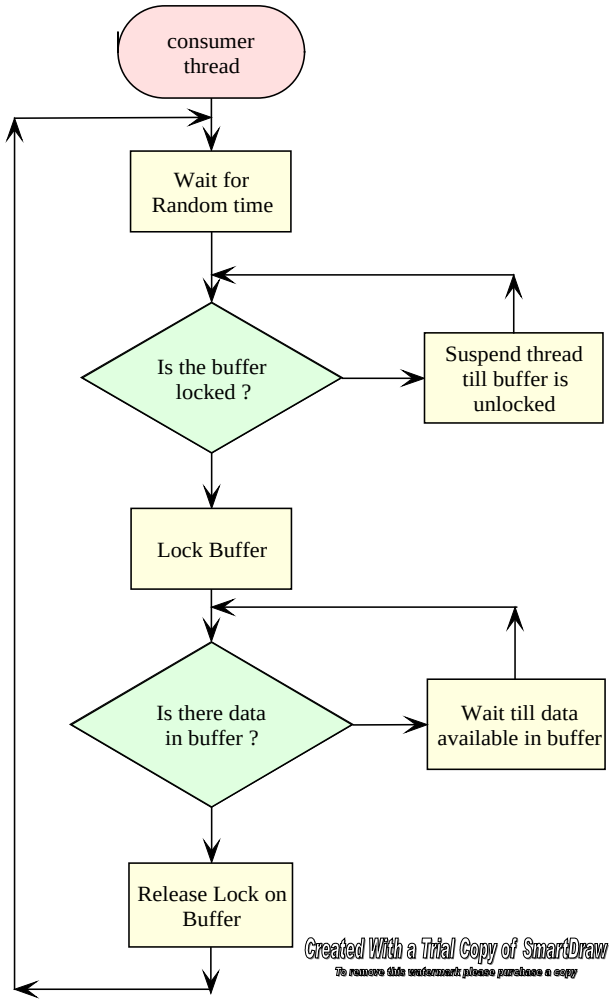
Producer/Consumer Flowchart for Method Based Implementation



(B) Flowchart for Producer thread



(C) Flowchart for Consumer thread



Code snippet

```
int sc_main(int argc, char *argv[])
{
        prod_cons *p1;
        p1 = new prod_cons("prod_cons_p1");
        sc_start (50, SC_NS);
        cout <<"Finished at time " << sc_time_stamp() << endl ;
        return 0;
}
```

```
#include "systemc.h"

SC_MODULE( prod_cons )
{
        void producer();
```

```
        void consumer();

        sc_fifo<sc_uint<20> > buffer;  // define buffer
        sc_mutex fifo; // mutex for the buffer

        // declare the output text file here
        ofstream outfile;

        int i;

        SC_HAS_PROCESS(prod_cons);
        prod_cons(sc_module_name name):sc_module(name)
        {
        outfile.open("prod_cons.txt");

        SC_THREAD(producer);
        SC_THREAD(consumer);

        // bring buffer to medium occupancy ..
        for (i=0;i<10;i++) buffer.write(i);
        }
};
```

```
#include "prod_cons.h"

// producer thread
void prod_cons::producer()
{
        int data;
        srand(40);
        data = 0;
        while(1) //loop so process is infinite
        {
                // wait for random amount of time before again
                // writing into the buffer
                wait ( ((rand() % 11) +1 ), SC_NS );

               // check if Buffer is locked ..
                fifo.lock();

                // check for room in the FIFO
                if (buffer.num_free() > 0)
                {

                        // generate sequential data
                        if (data >= 19) data = 0;
                        else data = data + 1;

                        // write data into the FIFO
                        buffer.write(data);

                        // Dump results in text file and screen
                cout << "\n\n At time" << sc_time_stamp() << "\n\t\t Data WRITTEN into the
buffer is "<< data ;
```

```
                    outfile << "\n\n At time "<< sc_time_stamp() <<" WRITE TRANSACTION: Data
written is" << data;
                    }
                    else
                    {
                        // fifo full condition
                        cout << "\n\n FIFO is Full !!" ;
                        outfile << "\n\n At time "<< sc_time_stamp()<<"FIFO is Full !!" ;  }

                        //release mutex
                        fifo.unlock();        //release mutex
                    }
        }
}

// consumer thread
void prod_cons::consumer()
{
        int read_data;
        srand(60);
        read_data = 0;

        while(1) //loop so process is infinite
        {
                // wait for random amount of time before again
                // reading from the buffer
                 wait ( ((rand() % 12)+1), SC_NS );

                 // check if Buffer is locked ..
                fifo.lock();

                // check for data in the FIFO
                if (buffer.num_available() > 0)
                {

                        // read data from the FIFO
                        read_data = buffer.read();

                        // Dump results in text file and screen
                        cout <<"\n\n At time "<< sc_time_stamp()<<"\n\t\t Data READ out is "<<
read_data ;
                        outfile <<"\n\n At time "<< sc_time_stamp() <<" READ TRANSACTION:
Data read out is"<< read_data ;
                }
                else
                {

                        // fifo empty condition
                        cout << "\n\n At time "<<sc_time_stamp() <<" FIFO is Empty !!" ;
                        outfile << "\n\n At time "<<sc_time_stamp() <<" FIFO is Empty !!" ;
                }

                //release mutex
                fifo.unlock();
        }
}
```

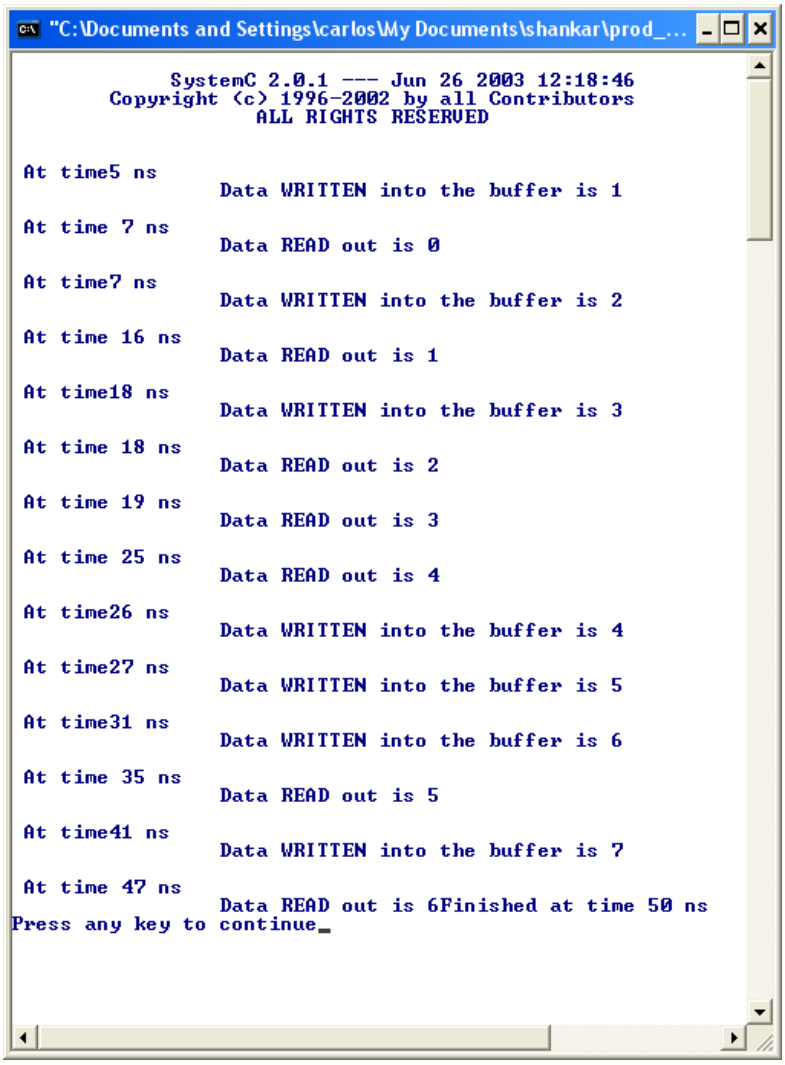


Figure 26 – Producer/Consumer: screen output

4. Evaluation

5. Observations

REFERENCES

CSI, 2004, our center's web site, http://www.csi.fau.edu/systemc_examples/

Username: systemc_user, Password: systemcfau

I have reproduced references from another paper. Arrange the remainder alphabetically. Change the font to be the same.

OSCI, 2002 *— Add the reference*

GROTKER, T., LIAO, S., MARTIN, G., AND SWAN, S., *System Design with SystemC,* Kluwer Academic Publishers, Boston, MA, 2002

BHASKER, J., *A SystemC Primer,* Star Galaxy Publishing, Allentown, PA, 2002

ITRS, *International Technology Roadmap for Semiconductors 2001 Edition,* Semiconductor Industry Association, ITRS, Austin, TX, 2001.

JAYADEVAPPA, S., AND SHANKAR, R., CAD Based Design Course Using a State of the Art System Level Language, *American Society for Engineering Education (ASEE),* Salt Lake City, Utah, June 2004, Session 262, Course and Curriculum Innovations, Paper No. 1.

SHANKAR, R., and JAYADEVAPPA, S., A New SystemC-based Foundation for the CE Curriculum, *European Workshop on Microelectronics Education (EWME)*, Lausanne, Switzerland, April 2004, pp. 33-37.

QURAISHI, G., AND SHANKAR, R., On simulating the IP Market Dynamics in an Academic Environment Using SystemC, *MSE (Microelectronics Systems Education) conference*, San Jose, CA, June 2003, pp.