SystemC Lab Exercises for Virtual Prototype elective course

Version 0.8

Table of Contents

1 C++ Lab Exercises

1.1 Bank Account

Aim of the Experiment

Design a class named *account*, *saving_account* & *current_account* as below. It should satisfy the test cases(TC) mentioned in the main function.

Template code

```
class account {};

class saving_account {};

class current_account {};

double fixed_deposit_rate(account *ac) {
    return (ac->type() == account::TYPE::SAVING) ? 6.5 : 4.0;
}

void delete_account(account *ac) { delete ac; }

void print_saving_account(account *ac) {
    if (/* type casting condition to print saving account only */) // Do not
use ac->type() method
    {
        cout << "holder : " << ac->holder() << endl;
        cout << "balance : " << ac->balance() << endl;
        cout << "interest rate : " << ac->interest_rate() << endl;
        cout << "FD rates : " << fixed_deposit_rate(ac) << endl;
    }
}

int main() {
    saving_account *s_ac = new saving_account("John");
    s_ac->deposit(1000);
    s_ac->withdraw(100.30);
    print_saving_account(s_ac);
    delete_account(s_ac);
    cout << endl;

    current_account *c_ac = new current_account("Mathew");
    c_ac->deposit(10);
    c_ac->withdraw(100.30);
    print_saving_account(c_ac);
    delete_account(c_ac);
    cout << endl;

    return EXIT_SUCCESS;
}
```

Expected Output

```
holder : John
balance : 899.7
interest rate : 4.5
```

<table>
<tr>
<td></td>
<td>

```
FD rates : 6.5
saving account of John deleted

current account of Mathew deleted
```

</td>
</tr>
<tr>
<td><b>Key Learnings</b></td>
<td>Inheritance
Virtual Method
Virtual Destructor
Enumerated types
C++ typecasting</td>
</tr>
</table>

1.2 Add to Output

<table>
<tr>
<td><b>Aim of the Experiment</b></td>
<td>To print the integer on output we generally use a statement like `cout << 5;`. Make `cout + 5;` statement to work as ` cout << 5;`.</td>
</tr>
<tr>
<td><b>Template code</b></td>
<td>

```
int main() {
    // TC 1
    cout + 5;

    // TC 2
    cout + 5.5f + "\nThis Works!";
    return EXIT_SUCCESS;
}
```

</td>
</tr>
<tr>
<td><b>Expected Output</b></td>
<td>

```
55.5
This Works!
```

</td>
</tr>
<tr>
<td><b>Key Learnings</b></td>
<td>Operator overloading
How printing(i.e. cout statement) works internally in C++.</td>
</tr>
</table>

1.3 Summable Array

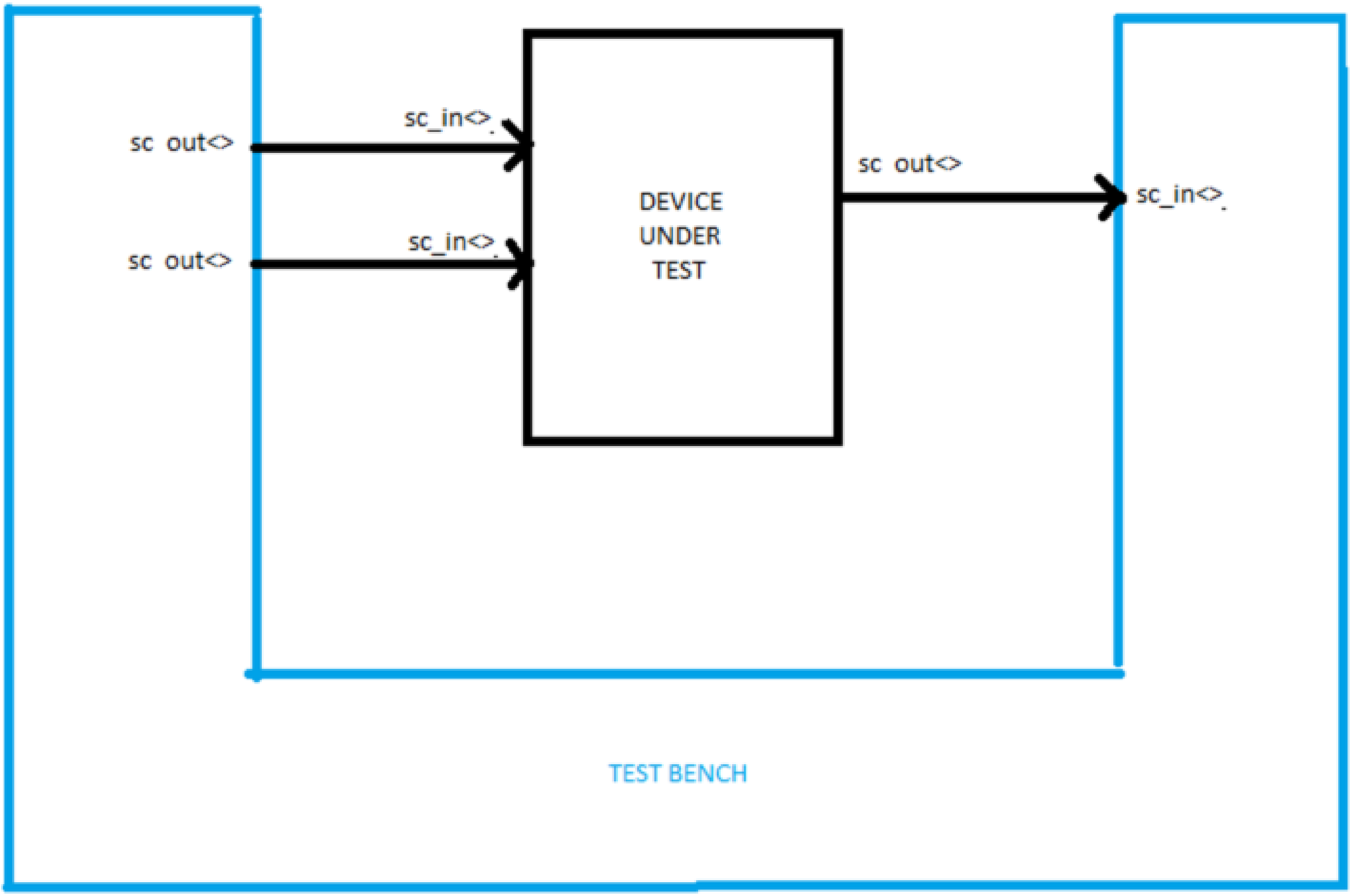
Aim of the Experiment	Design a class named array which works exactly like a primitive array. Except it has following extra functionality:

Provided code	class array {
	
	};
	
	int main() {
	    // TC 1
	    array<uint32_t, 5> arr1 = {1, 2, 3, 4, 5};
	
	    // TC 2
	    arr1[0] = 0;
	    assert(arr1[0] == 0);
	
	    // TC 3
	    array<uint32_t, 5> arr2 = {6, 7, 8, 9, 10};
	    array<uint32_t, 10> arr3 = arr1 + arr2;
	    assert(arr3.size() == 10);
	    assert((arr3 == array<uint32_t, 10>{0, 2, 3, 4, 5, 6, 7, 8, 9, 10}));
	
	    // TC 4
	    array<uint32_t, 5> arr4;
	    arr4 = arr2 = arr1;
	    assert((arr4 == array<uint32_t, 5>{0, 2, 3, 4, 5}));
	    assert((arr2 == array<uint32_t, 5>{0, 2, 3, 4, 5}));
	
	    return EXIT_SUCCESS;
	}

Expected Output	Code compiles & assert statements pass
Key Learnings	Operator (subscript, == & +) overloading Initializer list Template & Template Specialization Assignment operator Destructor

2 SystemC Lab Exercises

- All models should be implemented similar to the shown figure below.
- Device Under Test (DUT): Module developed
- Test Bench: Module create to configure DUT, route data over interface if needed. By definition, the Test Bench is a mirror model i.e., an sc_in in the DUT will have to be bound to sc_out in the Testbench.



2.1 Logic Gates

Objective	Implement two input gates (NAND, NOR, AND, OR, XOR, XNOR) using SystemC
Requirements	Input Ports ➢ *Boolean inputA*, Boolean *inputB* Output Ports ➢ Boolean *output*
Test Scenario	Use a testbench to update the input values and route the output to the testbench back to verify the behaviour of the DUT At time t, update values using *inputA* and *inputB* 0 ns : A = false, B = false 2 ns : A = false, B = true

	4 ns : A = true, B = false 6 ns : A = true, B = true
Pass Criteria	When the output matches respective gate model characteristic table as per give inputs (inputA, inputB) at every loop iteration
Key Learnings	Test focussed on SystemC basics: ➢ Provide Module definition: ○ SC_MODULE ○ class sc_module ➢ sc_main ➢ sc_start ➢ sc_end ➢ Use of boolean clock input port driven by sc_clock ➢ Input and Output ports ○ sc_in ○ sc_out ➢ SystemC Process: ○ SC_METHOD or SC_THREAD ➢ Data types ➢ Use of sc_signal ➢ Port binding ➢ Signal tracing with sc_trace ➢ Testbench to DUT communication
Comments	For more understanding on Logical gates functionality please refer this

2.2 Half Adder

Objective	Implement half adder adder using SystemC
Requirements	Input Ports ➢ *Boolean inputA*, Boolean *inputB* Output Ports ➢ Boolean *outputSum*, Boolean *outputCarry*
Test Scenario	1. At time t, update values using *inputA* and *inputB* 2. Observe the sum value reflected in *outputSum*, *outputCarry* 3. Increment loop count 4. If loop count is less than 4, at time t + delta, repeat above step #1 5. If loop count is 5, exit the test

Pass Criteria	When the *outputSum*, *outputCarry* matches half adder characteristic table as per given inputs (*inputA*, *inputB*) at every loop iteration
Key Learnings	Same as Exercise 2.1
Comments	For more understanding on Half Adder functionality please refer this

2.3 Full Adder

Objective	Implement full adder adder using SystemC
Requirements	Input Ports ➢ Boolean *inputA*, Boolean *inputB,* Boolean *inputC* Output Ports ➢ Boolean *outputSum*, Boolean *outputCarry*
Test Scenario	1. At time t, update values using *inputA* and *inputB*, *inputC* 2. Observe the sum value reflected in *outputSum*, *outputCarry* 3. Increment loop count 4. If loop count is less than 15, at time t + delta, repeat above step #1 5. If loop count is 16, exit the test
Pass Criteria	When the *outputSum*, *outputCarry* matches half adder characteristic table as per given inputs (*inputA*, *inputB, inputC*) at every loop iteration
Key Learnings	Same as Exercise 2.1
Comments	For more understanding on Full Adder functionality please refer this

2.4 Decoder

Objective	Implement 2:4 Decoder using SystemC
Requirements	Input Ports ➢ Boolean *inputA0* ➢ Boolean *inputA1* ➢ Boolean *EnableInput* Output Ports ➢ Boolean *Y0* ➢ Boolean *Y1* ➢ Boolean *Y2* ➢ Boolean *Y3*
Test Scenario	1. At time t, update values using *A1,A0* with *EnableInput* = 1 2. Observe the Output value reflected in *Y3,Y2,Y1,Y0* 3. Increment loop count 4. If loop count is less than 4, at time t + delta, repeat above step #1 5. If loop count is 5, exit the test 6. Repeat the above steps with *EnableInput* = 0, with *EnableInput* = 0, Output should to be in no change state
Pass Criteria	1. When EnableInput = 0, Output has to be all zero's irrespective of input. 2. When EnableInput = 1, One of the output signals has to HIGH based on the A1 and A0 input value
Key Learnings	Same as Exercise 2.1 Enable and Disabling of the module
Comments	For more understanding on Decoder functionality please refer this

2.5 Encoder

Objective	Implement 4:2 Encoder using SystemC
Requirements	Input Ports ➢ Boolean *Y0* ➢ Boolean *Y1* ➢ Boolean *Y2* ➢ Boolean *Y3* ➢ Boolean *EnableInput* Output Ports ➢ Boolean *inputA0* ➢ Boolean *inputA1,*

Test Scenario	1. At time t, update values using *Y3,Y2,Y1,Y0* with *EnableInput* = 1 2. Observe the Output value reflected in *A1,A0* 3. Increment loop count 4. If loop count is less than 15, at time t + delta, repeat above step #1 5. If loop count is 16, exit the test 6. Repeat the above steps with *EnableInput* = 0, with *EnableInput* = 0, Output should to be in no change state
Pass Criteria	3. Output A1,A0 value should be the index of the input index which is HIGH as that time 4. In 4-input lines, one input-line has to be set to true at any time to get the respective binary code in the output side
Key Learnings	Same as Exercise 2.1 Enable and Disabling of the module
Comments	For more understanding on Encoder functionality please refer this

2.6 Multiplexer

Objective	Implement 4:1 Multiplexer using SystemC
Requirements	Input Ports ➢ Boolean *inputA* ➢ Boolean *inputB* ➢ Boolean *inputC* ➢ Boolean *inputD* ➢ Boolean *inputCtrl0* ➢ Boolean *inputCtrl1* Output Ports ➢ Boolean *outMux*
Test Scenario	1. At time t, update values using *inputA* and *inputB, inputC* and *inputD* For all combinations of Ctrl lines, observe the value as selected by *inputCtrl0* and *inputCtrl1* reflected in *outMux* 2. Increment loop count 3. If loop count is less than 10, at time t + delta, repeat above step #1 4. If loop count is 10, exit the test
Pass Criteria	When the value at outMux matches the input line selected by inputCtrlx lines at every loop iteration

Key Learnings	Same as Exercise 2.1
Comments	For more understanding on Multiplexer functionality please refer this

2.7 D Flipflop

Objective	Implement D flipflop using SystemC
Requirements	Input Ports ➢ Boolean *D* ➢ Boolean *Clock* Output Ports ➢ Boolean *Q* ➢ Boolean *Qbar*
Test Scenario	1. At time t, update values using D, clock Observe the output value reflected in Q, Qbar 2. Increment loop count 3. If loop count is less than 10, at time t + delta, repeat above step #1 4. If loop count is 10, exit the test
Pass Criteria	1. When Q, Qbar matches D flip flop characteristic table as per given inputs (D, clock) at every loop iteration 2. The output should match the required characteristic table at clock edge but not at any other time instant other than clock edge (either pos or neg)
Key Learnings	**Use of boolean clock input port driven by sc_clock**
Comments	For more understanding on D Flipflop functionality please refer this

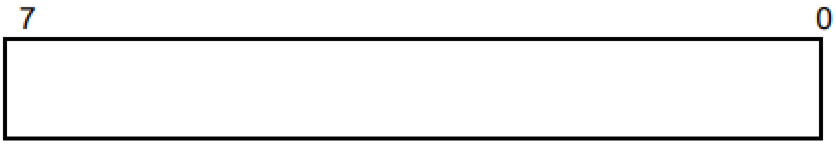
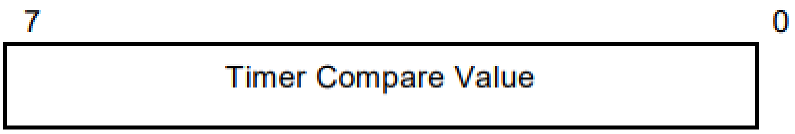
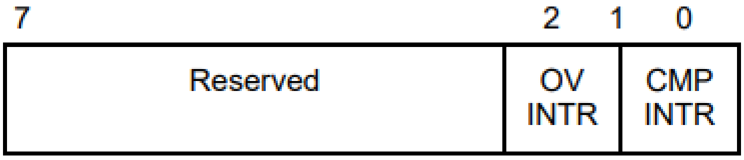
2.8 T Flipflop

Objective	Implement T flipflop using SystemC
Requirements	Input Ports ➢ Boolean *T* ➢ Boolean *Clock* Output Ports ➢ Boolean *Q* ➢ Boolean *Qbar*
Test Scenario	1. At time t, update values using *T*, *clock* Observe the output value reflected in *Q, Qbar* 2. Increment loop count 3. If loop count is less than 10, at time t + delta, repeat above step #1

	4. If loop count is 10, exit the test
Pass Criteria	When *Q, Qbar* matches T flip flop characteristic table as per given inputs (T, *clock*) at every loop iteration
Key Learnings	**Use of boolean clock input port driven by sc_clock**
Comments	For more understanding on D Flipflop functionality please refer this

2.9 8-bit Timer Counter Module – Cycle Accurate

<table>
<tr><td><b>Objective</b></td><td>Implement 8-bit Timer Counter Module using SystemC</td></tr>
<tr><td><b>Requirements</b></td><td>
Input Ports
<ul>
<li>Boolean <i>clock</i></li>
<li>Boolean <i>reset</i></li>
<li>Uint <i>address&lt;8&gt;</i></li>
<li>Uint data&lt;8&gt;</li>
<li>Boolean <i>read_enable</i></li>
<li>Boolean <i>write_enable</i></li>
</ul>
Output Ports
<ul>
<li>Boolean <i>interrupt0</i> (Active High Pulse Interrupt)</li>
<li>Boolean <i>interrupt1</i> (Active High Pulse Interrupt)</li>
</ul>
<p>Registers
The module should also consists of couple of 8bit Special Function Registers(SFRs) to configure the Timer module. The register sets as defined as below:</p>
<p>1. Timer control Register(CTRL) - Offset Address = 0x0</p>
<table><tr><td>Reserved</td><td>OV</td><td>CMP</td><td>EN</td></tr></table>
<p>0bit - En - Timer Enable bit, where if you configure it to:
0 => Timer is disable and counter does not increment All register can still be read and written
1 => Timer is enabled and counter will increment normally. Timer counter will be incremented every clock cycle.</p>
<p>1bit - CMP- Timer compare Interrupt Enable, where if you configure it to:
0 =>Compare interrupt is disabled
1 => Compare interrupt is enable.
If enabled then whenever timer counter value reaches to compare value it will raise the interrupt line(Interrupt0).</p>
<p>2bit - OV - Overflow Interrupt Enable, where if you configure it to:
0 =>Overflow interrupt is disabled
1 => Overflow interrupt is enable.
if timer counter value reaches to OxFF, in the next clock cycle it will update start counting from zero, and raise the interrupt line(Interrupt1) corresponding to overflow interrupt.</p>
</td></tr>
</table>

	2. Timer Value Register - Offset Address = 0x4  This is a read only register, Whenever timer is enabled it will start incrementing value of this register. 3. Timer Compare Register - Offset Address = 0x8  This register can be used to store the comparison value for timer counter register. If timer compare interrupt is enable, then it will assert an interrupt line INTO whenever timer value matches with the timer compare value. 4. Timer Interrupt status Register - Offset Address = 0xC  This is a write one clear interrupt status register. If timer is enabled then whenever timer counter value reaches a comparator value or an overflow occurs then it will update the corresponding status bit if related interrupt is enabled. Writing a 1 will clear the corresponding interrupt
Test Scenario	1. This is an 8-bit incrementing counter. With 2 interrupts out which are used to represent the compare match and overflow of counter status 2. It has a private 8-bit comparator that is used to assert an interrupt (*interrupt0*) when the timer has reached the comparator value. 3. Overflow event, i.e., whenever timer value reaches OxFF, in the next increment counter will start counting from Ox0 and raise an interrupt line(*interrupt1*) corresponding to overflow interrupt. 4. The timer is clocked by CLK. Negative Test Scenario: 5. The *interrupt0*/1 has to be logic low in entire simulation time except when overflow or comparator match event is triggered
Pass Criteria	1. Registers set should be accessible from TB 2. Interrupt0 and Interrupt1 has to be triggered at expected time and also needs to update corresponding status register bit fields
Key Learnings	**1. Register implementations, configuration and update**

	2. Raising interrupts from module 3. **Cycle Accurate modelling**
Comments	For more understanding on Timer/Counter functionality please refer this and for register interface refer this

2.10 8-bit Timer Counter Module – Event Based

<table>
<tr><td><b>Objective</b></td><td>Implement Event based 8 bit Timer Counter Module using SystemC</td></tr>
<tr><td><b>Requirements</b></td><td>Input Ports
➢ sc_time <i>inputClock</i> (This will be configured with clock time period at which it should increment its counter and this time period value is provided by TB)
➢ Boolean <i>reset</i>
➢ Uint <i>address&lt;8&gt;</i>
➢ Uint data<8>
➢ Boolean <i>read_enable</i>
➢ Boolean <i>write_enable</i>
Output Ports
➢ Boolean <i>interrupt0</i> (Active High Pulse Interrupt)
➢ Boolean <i>interrupt1</i> (Active High Pulse Interrupt)

Registers
The module should also consists of couple of 8bit Special Function Registers(SFRs) to configure the Timer module. The register sets as defined as below:

2. Timer control Register(CTRL) - Offset Address = 0x0
<table><tr><td>Reserved</td><td>OV</td><td>CMP</td><td>EN</td></tr></table>
0bit - En - Timer Enable bit, where if you configure it to:
0 => Timer is disable and counter does not increment All register can still be read and written
1 => Timer is enabled and counter will increment normally. Timer counter will be incremented every clock cycle.

1bit - CMP- Timer compare Interrupt Enable, where if you configure it to:
0 =>Compare interrupt is disabled
1 => Compare interrupt is enable.
If enabled then whenever timer counter value reaches to compare value it will raise the interrupt line(Interrupt0).

2bit - OV - Overflow Interrupt Enable, where if you configure it to:
0 =>Overflow interrupt is disabled
1 => Overflow interrupt is enable.
if timer counter value reaches to OxFF, in the next clock cycle it will update start counting from zero, and raise the interrupt line(Interrupt1) corresponding to overflow interrupt.</td></tr>
</table>

	3. Timer Value Register - Offset Address = 0x4 7 … 0 This is a read only register, Whenever timer is enabled it will start incrementing value of this register. 4. Timer Compare Register - Offset Address = 0x8 7 … 0: Timer Compare Value This register can be used to store the comparison value for timer counter register. If timer compare interrupt is enable, then it will assert an interrupt line INTO whenever timer value matches with the timer compare value. 5. Timer Interrupt status Register - Offset Address = 0xC 7 … 2: Reserved; 1: OV INTR; 0: CMP INTR This is a write one clear interrupt status register. If timer is enabled then whenever timer counter value reaches a comparator value or an overflow occurs then it will update the corresponding status bit if related interrupt is enabled. Writing a 1 will clear the corresponding interrupt
Test Scenario	Implement the 8-bit Timer Counter Module with event-based approach: 1. The timer model should not have the pin for the clock. 2. The timer should not count/increment at every clock edge, rather we should apply the mathematical formulas to calculate when the interrupts will be generated. that means increment should not be sensitive to the clock. 3. The testbench should remain the same, only the clock pin will be removed. 4. You should get exactly the same output, which means functionality will be the same. 5. Note down how much wall clock time is consumed by the simulation. 6. Run both implementations of models, and check the difference in the wall clock time elapsed for simulating the same amount of simulation time. And conclude your decision. **Note: Timer module has to clear the interrupts (*interrupt0* and *interrupt1*) (once it is set to due to trigger condition) after one cycle of the input time interval**

	Negative Test Scenario: 1. The Interrupt0/1 has to be logic low in entire simulation time except when overflow or comparator match event is triggered
Pass Criteria	1. Modelling the timer module by using sc_time and not the bool clock 2. Registers set should be accessible from TB 3. Interrupt0 and Interrupt1 has to be triggered at expected time and also needs to update corresponding status register bit fields
Key Learnings	**1. Event Based Modelling** **2. Raising/Toggling the Interrupts using input sc_time clock period as reference (Without Boolean clock)**
Comments	For more understanding on Timer/Counter functionality please refer this and for register interface refer this (This is just for general understanding purpose not for the scope of modelling)

2.11 Memory Module

Objective	Implement 1KB of Memory module using SystemC
Requirements	Input Ports: ➢ Unsigned int *address*<32> Boolean *read_or_write_en* (0 -> read, 1 -> write) Inout Ports: ➢ Unsigned int *data*<32> Develop the memory model (Size: 1 KB) by using dynamic memory allocation Assume it has 32-bit data lines, 32-bit address lines, *read_en / write_en line, reset_n* line
Test Scenario	In the testbench, do the following: 1. Read the 1 KB of data from a file and store in internal buffer buff1 2. Start the following at 10 ns 3. Write this entire data from the testbench to memory model 4. Read the entire data from memory model to testbench and store in another temporary buffer buff2 5. Compare buff1 & buff2
Pass Criteria	1. Memory module should be able to read and write 2. buff1 and buff2 content should match
Key Learnings	**1. Memory implementation** **2. Read and write to memory** 3. **Inout interface**

Comments	

2.12 Hello world example using TLM 2.0

Objective	Implement simple Hello world example using TLM 2.0 concepts using b_transport call
Requirements	Initiator Module Interface (TB) ➢ tlm_initiator_socket <> ***InitSocket_*** Target Module Interface (DUT) ➢ tlm_target_socket<> ***TargetSocket_*** Loosely Timed(LT)/Blocking interface (b_transport) has to be supported by target module
Test Scenario	1. ***InitSocket_*** of Initiator module and ***TargetSocket_*** of Target module has to be binded using bind method 2. Initiator will make a call to b_transport with some delay 3. Target side calls a method that will print "Hello World" whenever there is a message from the Initiator 4. Optionally perform "wait" of said delay time units in the target
Pass Criteria	Perform a call from initiator socket and print "Hello world" as the call reaches the target socket
Key Learnings	**1. TLM 2.0 initiator socket** **2. TLM 2.0 target socket** **3. b_transport**
Comments	For more understanding on sockets and Blocking transport (b_tansport) please refer this (This is just for general understanding purpose not for the scope of modelling exercise)

2.13 8-bit Timer Counter Module with TLM2.0 sockets

Objective	Implement Event based 8-bit Timer Counter Module using TLM2.0 Sockets i.e., (TLM Initiator Sockets and TLM Target Sockets)
Requirements	Input Ports ➢ sc_time ***inputClock*** (This will be configured with clock time period at which it should increment its counter and this time period value is provided by TB) ➢ Boolean *reset* ➢ Boolean *read_enable* ➢ Boolean *write_enable*

- tlm_target_socket<> ***sfrTargetSocket***

Output Ports

- Boolean *interrupt0* (Active High Pulse Interrupt)
- Boolean *interrupt1* (Active High Pulse Interrupt)

sfrTargetSocket has to be used to configure and access the Timer/Counter SFR registers using b_transport calls

Registers

The module should also consists of couple of 8bit Special Function Registers(SFRs) to configure the Timer module. The register sets as defined as below:

3. Timer control Register(CTRL) - Offset Address = 0x0

Reserved	OV	CMP	EN

0bit - En - Timer Enable bit, where if you configure it to:
0 => Timer is disable and counter does not increment All register can still be read and written
1 => Timer is enabled and counter will increment normally. Timer counter will be incremented every clock cycle.

1bit - CMP- Timer compare Interrupt Enable, where if you configure it to:
0 =>Compare interrupt is disabled
1 => Compare interrupt is enable.
If enabled then whenever timer counter value reaches to compare value it will raise the interrupt line(Interrupt0).

2bit - OV - Overflow Interrupt Enable, where if you configure it to:
0 =>Overflow interrupt is disabled
1 => Overflow interrupt is enable.
if timer counter value reaches to OxFF, in the next clock cycle it will update start counting from zero, and raise the interrupt line(Interrupt1) corresponding to overflow interrupt.

4. Timer Value Register - Offset Address = 0x4

7	0

This is a read only register, Whenever timer is enabled it will start incrementing value of this register.

5. Timer Compare Register - Offset Address = 0x8

7	0
Timer Compare Value	

<table>
<tr><td></td><td>This register can be used to store the comparison value for timer counter register. If timer compare interrupt is enable, then it will assert an interrupt line INTO whenever timer value matches with the timer compare value.
6. Timer Interrupt status Register - Offset Address = 0xC
<table><tr><td>7</td><td>2</td><td>1</td><td>0</td></tr><tr><td>Reserved</td><td></td><td>OV INTR</td><td>CMP INTR</td></tr></table>
This is a write one clear interrupt status register. If timer is enabled then whenever timer counter value reaches a comparator value or an overflow occurs then it will update the corresponding status bit if related interrupt is enabled. Writing a 1 will clear the corresponding interrupt
<b>NOTE: Timer module should support for b_transport interface through which registers access are handled.</b>
Testbench related info:
1. Testbench Interface
➢ tlm_initiator_socket<> <b><i>tbInitiatorSocket</i></b>
2. TB has to access the Timer module through its initiator socket to target socket of Timer module
3. All Timer module register access by TB has to be done by b_transport call and transaction related info (such as read/write command, address, data pointer) should be passed using tlm_generic_payload</td></tr>
<tr><td><b>Test Scenario</b></td><td>Implement the 8-bit Timer Counter Module with event-based approach:
1. The timer model should not have the pin for the clock.
2. The timer should not count/increment at every clock edge, rather we should apply the mathematical formulas to calculate when the interrupts will be generated. that means increment should not be sensitive to the clock.
<b>3. Timer modules registers should be accessed via b_transport calls only</b>
4. The testbench should remain the same, only the clock pin will be removed.
5. You should get exactly the same output, which means functionality will be the same.
6. Note down how much wall clock time is consumed by the simulation.
7. Run both implementations of models, and check the difference in the wall clock time elapsed for simulating the same amount of simulation time. And conclude your decision.
Negative Test Scenario:</td></tr>
</table>

	1. The Interrupt0/1 has to be logic low in entire simulation time except when overflow or comparator match event is triggered
Pass Criteria	1. Model the timer module by using sc_time and not the bool clock 2. Registers set should be accessible from TB via b_transport call 3. Interrupt0 and Interrupt1 has to be triggered at expected time and also needs to update corresponding status register bit fields
Key Learnings	1. TLM 2.0 initiator socket 2. TLM 2.0 target socket **3. tlm_generic_payload usage**
Comments	For more understanding on Timer/Counter functionality please refer this and for register interface refer this (This is just for general understanding purpose not for the scope of modelling) **Optional**: **Update the same model to support Fastly Timed (FT) Interface/Non-Blocking (nb_transport) to configure the Timer SFRs via TB, just remember in such case TB should also needs to support FT interface i.e., backward path methods**

2.14 Memory Module using TLM2.0 Sockets

Objective	Implement 1KB of Memory module using TLM2.0 Sockets i.e., (TLM Target Sockets for Memory module access) via b_transport, transport_dbg and get_direct_mem_ptr call
Requirements	Memory Module Interfaces: ➢ tlm_target_socket<> *mem_target_socket* Develop the memory model (Size: 1 KB) by using dynamic memory allocation Implement Forward path methods in Memory module i.e., Loosely Timed interface - **b_transport**, Debug Interface - **transport_dbg** DMI - **get_direct_mem_ptr** Testbench Inetrface- ➢ tlm_initiator_socket<> *tb_initiator_socket* TB has to access i.e., read and write of Memory module through Initiator socket of TB and target socket of Memory by having b_transport and transport_dbg calls
Test Scenario	In the testbench, do the following: 1. Read the 1 KB of data data from a file and store in internal buffer buff1 2. Start the following at 10 ns

	3. Write this entire data from the testbench to memory model **by using b_transport calls** 4. Read the entire data from memory model to testbench and store in another temporary buffer buff2 5. Compare buff1 & buff2 Repeat the above steps using Debug interface(transport_dbg call) and DMI interface (get_direct_mem_ptr call) as well
Pass Criteria	In all of the three interfaces i.e., LT, Debug and DMI: 1. Memory module should be able to read and write 2. buff1 and buff2 content should match
Key Learnings	1. LT Interface (b_transport) **2. Debug Interface (transport_dbg)** 3. **DMI (get_direct_mem_ptr)**
Comments	**Optional**: **Update the same model to support Fastly Timed (FT) Interface/Non-Blocking (nb_transport) to access Memory Module via TB, just remember in such case TB should also needs to support FT interface i.e., backward path methods**

2.15 UART Serial Communication Module

Objective	Implement simple UART communication Module to transmit and receive data serially
Requirements	UART Interfaces and Ports: ➢ tlm_target_socket<> *uart_sfr_target_socket* ➢ Boolean uart_tx ➢ Boolean uart_rx Implement Loosely Timed interface (**b_transport)** in UART module to access the UART SFR registers Testbench Inetrface- ➢ tlm_initiator_socket<> *tb_initiator_socket* ➢ Boolean uart_rx ➢ Boolean uart_tx UART Register Set (All are 32-bit registers and Default/Reset Value of registers is 0x0) 1. Control Register (CTRL) – Offset Address = 0x0 ➢ Bit0 – Enable ○ 0 = UART module is disabled (Transmission and reception is disabled) ○ 1 = UART module is disabled (Transmission and reception is disabled)

	➢ Bit1 to Bit5 – Data Format Length ○ Provides the length of data which needs to be sent/receive in one data packet ➢ Bit6 – Start Transfer ○ 1 = Starts the transmission of data configured in Transmit_Data_Register(of length and baudrate provided in CTRL register) ➢ Bit7 to Bit15 - Reserved ➢ Bit16 to Bit31 – Baudrate ○ Provides the rate at which data has to be sent or received 2. Received Data Register(RxData Reg) – Offset Address = 0x4 (Read only register) ➢ Bit0 to Bit15 – *Data* ○ Last data that UART module received via uart_rx ➢ Bit16 to Bit31 – Reserved 3. Transmit Data Register (TxData Register) – Offset Address = 0x8 ➢ Bit0 to Bit15 – *Data* ○ Data that needs to transmitted from UART module via uart_tx (at length and baudrate configured in CTRL register) ○ Data is will be transmitted only when CTRL.START_TRANSFER bit is set ➢ Bit16 to Bit31 - Reserved
Test Scenario	In TB do the following steps: 1. Configure the UART modules SFR using b_transport call 2. Configure Data format length, Baudrate for the data transmission 3. Configure the data to be sent 4. Configure the UART to start the data transfer 5. Read the data received from UART.uart_tx out port via TB.uart_rx in port and check whether data received by TB is same as configured 6. Transmit some data from TB.tx to UART.uart_rx with length and baudrate configured in step2 7. Read the Received Data register and the data with data sent

Pass Criteria	UART module should be able to send and receive configured data properly with configured data format length and baudrate
Key Learnings	**Serial Communication Implementation**
Comments	