Lecture 4
Fundamentals of SystemC (Part II)

Multimedia Architecture and Processing Laboratory
多媒體架構與處理實驗室
Prof. Wen-Hsiao Peng (彭文孝)
pawn@mail.si2lab.org
2007 Spring Term
Acknowledgements

This lecture note is partly contributed by Prof. Gwo Giun Lee (李國君) in the Dept. of EE, National Cheng-Kung University and his team members 王明俊, 林和源 in the research laboratory 多媒體系統晶片實驗室

E-mail: clee@mail.ncku.edu.tw
Tel: +886-6-275-7575 ext. 62448
Web: http://140.116.216.53
References

Untimed TLM in SystemC
Untimed TLM – Data Flow Modeling

◆ Purposes
  ❖ Create initial draft of the specification model

◆ Focus
  ❖ Data flow and functionality modeling

◆ Kahn process network
  ❖ Deterministic, untimed model of computation

◆ How?
  ❖ Untimed processes are modeled with SC_THREAD
  ❖ Processes communicate through FIFO channels using blocking read/write
  ❖ Implicit system synchronization is achieved by the blocking read/write
    ✷ Blocking read is suspended until data is available in the FIFO channel
    ✷ Blocking write is suspended until more space is available in the FIFO channel
A Simple Adder

Behavioral modeling

Feedback with Delay

constant

adder

fork

printer

\( z^{-1} \)

sc_fifo, primitive channels
A Simple Adder (c. 1)

Implementation of blocking read/write includes wait statements

SC_THREAD is a must for accessing blocking read/write

Constant generator

#include <systemc.h>
#include <iostream>
template <class T> SC_MODULE(DF_Adder){
  sc_fifo_in<T> input1, input2;
  sc_fifo_out<T> output;
  void process()
  {
    while(1)
      output.write(input1.read()+input2.read());
  }
  SC_CTOR(DF_Adder) {SC_THREAD(process);} 
};
template <class T> SC_MODULE(DF_Const)
{
  sc_fifo_out<T> output;
  void process() {while (1) output.write(constant_);} 
  SC_HAS_PROCESS(DF_Const); 
  DF_Const(sc_module_name N, const T&C):
  {SC_THREAD(process);}
  T constant_; 
};
template <class T> SC_MODULE(DF_Fork)
{
  sc_fifo_in<T> input;sc_fifo_out<T> output1,output2;
  void process()
  {
    while(1) {
      T value=input.read();
      output1.write(value);
      output2.write(value);
    }
  }
  SC_CTOR(DF_Fork) {SC_THREAD(process);} 
};
template <class T> SC_MODULE(DF_Printer)
{
  sc_fifo_in<T> input;
  SC_HAS_PROCESS(DF_Printer);
  DF_Printer(sc_module_name NAME, unsigned N_ITER):
  {SC_THREAD(print_process);}
  ……See the next page……..
  
  SC_THREAD is a must for accessing blocking read/write
void print_process() {
    for (unsigned i = 0; i < n_iterations_; i++) {
        T value = input.read();
        cout << name() << " " << value << endl;
    }
    done_ = true;
    return;
}
~DF_Printer() {
    if (!done_) cout << name() << "not done yet" << endl;
}
unsigned n_iterations_;
bool done_;
**A Simple Adder (c. 3) – Deadlock**

- Missing initial values of the feedback channel
  - The adder would never be able to read from its first input
- Unbalanced production and consumption rate in a feedback loop
  - A producer and a consumer connected to a FIFO channel need to move the same number of tokens on the average
  - If “DF_fork” produces two values on each output for every data sample consumed, the simulation would stall with the folk module attempting to write to its first output port
- A quest for empty and full FIFOs can help to identify the causes
  - Access suspicious FIFOs after returning from sc_start()
A Simple Adder (c. 4) – FIFO Sizes

- FIFO sizes play a key role
  - Determine whether a system deadlocks
  - Have an impact on system behavior (value sequences)
  - Have an impact on simulation speed
- Larger sizes cause less context switches and faster simulation
Insertion of Functional Delay

- Inserting functional delay in untimed TLM can be achieved by annotating the delay with wait statements

```c
void process(){
    while(1)
    {
        T data = input1.read()+input2.read();
        wait(200, SC_NS);
        output.write(T);
    }
}
```

- SC_THREAD is more preferable than SC_METHOD
  - Adding delay information in method processes is nontrivial
  - Maintaining blocking read/write semantics in method processes is next to impossible
Stopping Untimed TLM Simulation

- Simulation time will never advance in a purely untimed TLM
  - Processes are simulated with advance of delta-cycles
  - Using `sc_start` with a positive argument does not lead to the termination of the simulation

- How can one terminate the simulation?
  - Solution 1
    - Use process termination and hope for data backlog
      - Simulations are run until a sufficient amount of samples are obtained
      - Example: the `print_process` of the `DF_printer` module
      - It is not guaranteed to be successful
        - The process may be in some dead branch of the system
    - There could be multiple instances of `DF_printer`
      - Logic and-combination of exit conditions is a must
Stopping Untimed TLM Simulation (c. 1)

- Solution II
  - Simply indicate the fact and continue to consume data
  - Create a terminator module and monitor different exit conditions
  - Combination of a given simulation time and data-dependent exit conditions can often be the best solution

```cpp
template <class T, unsigned n_iterations>
SC_MODULE(DF_Printer) {
sc_fifo_in<T> input;
sc_out<bool> done;
SC_CTOR(DF_Printer) {
SC_THREAD(print_process);
done.initialize(false);
}
void print_process() {
for(unsigned i=0; i<n_iterations_; i++) {
    cout << name() << " " << input.read() << endl;
}
done_ = true;
while(1) input.read();
}
}
```

```cpp
SC_MODULE(Terminator) {
sc_port<sc_signal_in_if<bool>, 0> inputs;
SC_CTOR(Terminator) {
    SC_METHOD(arnbold);
sensitive << inputs;
}
void arnold() {
    for(unsigned i=0; i<inputs.size(); i++) {
        if(inputs[i]->read() == false) return;
    }
    sc_stop();
}
```

The “inputs” port can be connected to an arbitrary number of Boolean signals.
Exercises

(1) Dump the execution order of the different processes
(2) Set the sizes of all the FIFO channels to 5, and compare the execution order of the processes with that in (1)
(3) Continue the experiment in (1) and modify the DF_fork module such that it produces two values on each output for every data sample consumed. Identify the step in which the simulation halts and state the reasons
(4) Continue the experiment in (3), break the feedback loop, remove the adder, and state whether the system will halt or not? Give the reasons behind your observations.
(5) Continue the experiment in (1), set the delay $z^{-1}$ to $z^{-3}$
Interface and Channel Design
Interface and Channel

- Interfaces and channels provide a flexibly way to model communication and system synchronization
  - Interface classes declare the access methods
  - Channels implement the access methods declared within interfaces
- Interfaces separate the communication from the computation
  - Communication refinement is achieved by allowing one channel to easily be swapped with another

\[ 
\text{M1} \quad \text{Channel 1} \quad \text{M2} \\
\text{M1} \quad \text{Channel 2} \quad \text{M2} \\
\]

- port
- interface
**Interface Design**

- Good interface design shall reduce the modeling effort, make design refinement easier, and increase design reuse

- Guidelines for designing interface classes
  - Minimize the number of distinctive interfaces
    - Allow the channel implementation be easily swapped
    - Leave the port instances untouched during the replacement of channels
  - Layer specialized interfaces on more general interfaces
    - Specialized interfaces inherit from generalized interfaces
  - Use class inheritance to group common interface
    - Create a common base class for different interface classes
  - Use multiple inheritance to create a unified interface class
    - Define the unified interface to inherit from separate interfaces without adding any new members
Layered Interface Classes

- Allow channel implement interfaces of different specialization

- General Interface
  - Specialized Interface
    - More Specialized Interface
      - Most Specialized Interface

Simple channel that implements only the methods in general interface

More complex channel that implements the methods in general, specialized, and more specialized interfaces
**Layered Interface Classes (c. 1) – Channel Reuse**

- Modules use the least specialized interface, but channels that implement more-specialized interfaces can still be attached.

```cpp
class simple_rw_if: virtual public sc_interface {
    public:
    virtual void read(unsigned addr, char* data) = 0;
    virtual void write(unsigned addr, char* data) = 0;
};

class burst_rw_if: public simple_rw_if {
    public:
    virtual void burst_read(unsigned addr, char* data, unsigned n) = 0;
    virtual void burst_write(unsigned addr, char* data, unsigned n) = 0;
};

class simple_module: public sc_module {
    public:
    sc_port<simple_rw_if> rw_port;
};
```

No need for change when attaching different channels.
**Channel Design**

- **Primitive channels**
  - Derived from `sc_prim_channel`
  - Contain no hierarchy and processes
  - May implement the request-update mechanism
  - Faster simulation speed

- **Hierarchical channels**
  - Derived from `sc_channel`
  - May contain hierarchy and processes
  - Slower simulation speed due to more processes and context switches

- What if your channel needs request-update mechanism and contains processes, ports, and modules?
  - Split the channel into a primitive channel and a hierarchical channel
  - Instance the primitive channel in the hierarchical one
Primitive Channels
**Primitive Channels with Request-Update**

- **Key ideas of request-update**
  - Gather state change requests during the *evaluation* phase
  - Determine the next state of the channel during the *update* phase
  - Propagate the new channel state during the next *delta-cycle*

- **Main uses**
  - Delta-cycle delay communication for hardware simulation
    - A new value assigned to a channel does not take effect immediately
    - There is always a delta-cycle delay before the new value takes effect
  - Arbitration and resolution of simultaneous actions
    - Simultaneous actions occur within the same simulation phase
    - Either is free to execute before the other
    - Example: multiple processes drive the same channel

- **Primitive channels with request-update in SystemC**
  - `sc_fifo`, `sc_signal`, `sc_buffer`, `sc_signal_resolved`, `sc_signal_rv`
**Primitive Channels with Request-Update (c. 1)**

**sc_signal**

```cpp
template <class T> class sc_signal :
    public sc_signal_inout_if<T>,
    public sc_prim_channel
{
    public:
    // get the value changed event
    virtual const sc_event& value_changed_event() const = 0;
    // read the current value
    virtual const T& read() const = 0;
};
```

```cpp
template <class T> class sc_signal_inout_if :
    public sc_signal_inout_if<T>
{
    public:
    // write the new value
    virtual void write( const T& ) = 0;
};
```

Any read in the current delta cycle will get the old value.

Inform the scheduler to call `update()` in the update phase of the current delta-cycle.

Trigger the read process in the next delta-cycle to read the new value.

**Shift Registers**

**Derived from sc_prime_channel**

**Model of Primitive Channel with Request-Update**
**Primitive Channels without Request-Update**

- **sc_mutex (mutual exclusion object)**
  - Allow multiple processes share a common resource

- **Usage**
  - The process using the resource locks the mutex and unlocks it afterwards when the resource is no longer needed
  - Other processes must wait for the resource to be freed
  - The process executing first after the unlock will succeed in locking the mutex
  - There is no guarantee about which process will succeed

- **Member functions**
  - **lock()**
    - Lock the mutex (wait until unlocked if in use)
  - **trylock()**
    - Non-blocking, return true if success, else false
  - **unlock()**
    - Free previously locked mutex

```cpp
class bus{
  sc_mutex bus_access;
  ...
  void write (int addr, int data) {
    bus_access.lock();
    // perform write
    bus_access.unlock();
  }
  ...
}; // end class
```
**Primitive Channels without Request-Update (c. 1)**

- Implementation of `sc_mutex`

```cpp
class sc_mutex: virtual public sc_interface
{
    public:
    //blocks until mutex could be locked
    virtual void lock () = 0;

    //returns false if mutex could not be locked
    virtual bool trylock () = 0;

    //unlocks mutex
    virtual void unlock()=0;
};
```

```cpp
class sc_mutex: public sc_mutex_if, public sc_prim_channel{
    public:
    virtual void lock() {
        while(_locked)
            wait(_free);
        _locked = true;
    }

    virtual bool trylock(){
        if(_locked) return false;
        _locked = true;
        return true;
    }

    virtual void unlock (){ 
        _locked = false;
        _free.notify();
    }
private:
    sc_event _free;
    bool _locked;
};
```

Processes must wait for the resources to be freed

Processes waiting to lock the mutex will be resumed via the immediate notification

The process that owns the resource needs to release the resource
**Primitive Channels without Request-Update (c. 2)**

- **sc_semaphore**
  
  - Allow multiple processes share a set of common resources
  - A generalized version of sc_mutex

- **Member functions**
  
  - **wait()**
    
    - If available, occupy one semaphore
    - Otherwise, suspend until there is an available resource
  
  - **trywait()**
    
    - If available, occupy one semaphore
    - Otherwise, return -1
  
  - **get_value()**
    
    - Return number of available semaphore
  
  - **post()**
    
    - Free the previous occupied semaphore

```cpp
class multiport_RAM {
  sc_semaphore read_ports (3);
  sc_semaphore write_ports (2);
  ...
  void read (int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }

  void write (int addr, int data) {
    write_ports.wait();
    // perform write
    write_ports.post();
  }
  ...
}
```

Specify the maximum number of processes that can be granted for resource access.
**Primitive Channels without Request-Update (c. 3)**

- Implementation of `sc_semaphore`

```cpp
class sc_mutex_if :
virtual public sc_interface {
    // Lock the semaphore, block if not available
    virtual int wait() = 0;

    // Lock the semaphore, return -1 if not available
    virtual int trywait() = 0;

    // Unlock the semaphore
    virtual int post() = 0;
};
```

```cpp
class sc_semaphore :
public sc_semaphore_if,
public sc_prim_channel{
public:
    virtual int wait() {
        while(in_use())
            wait( m_free );
        --m_value;
        return 0;
    }

    virtual int trywait() {
        if( in_use() )
            return -1
        -- m_value;
        return 0;
    }

    virtual int post() {
        ++ m_value;
        m_free.notify();
        return 0;
    }

private:
    bool in_use() const
        { return ( m_value <= 0 ); }

    sc_event m_free;
    int m_value;
};
```

Blocking wait

Non-blocking call

Release the resource

- Class `sc_mutex_if`:
  - virtual `wait` method
  - virtual `trywait` method
  - virtual `post` method

- Class `sc_semaphore`:
  -virtual `wait` method
  - virtual `trywait` method
  - virtual `post` method
  - protected: `in_use` method
  - private: `sc_event m_free`, `int m_value`
Hierarchical Channels
Hierarchical Channels

- User defined channel
- May have embedded modules, channels, and processes
- Provide implementations for one or more interfaces
  - A hierarchical channel is distinguished from a general module by the fact that it implements interfaces
- Used to encapsulate both the structural elements of a design and the communication protocol or methods
Channel Refinement

- Communication via primitive `sc_fifo` channel

- Communication via hierarchical channel containing RTL implementation
**Communication via SC_FIFO**

```cpp
class producer : public sc_module {
    public:
        sc_port<sc_fifo_out_if<char>> out;
        SC_HAS_PROCESS(producer);
        producer(sc_module_name name) : sc_module(name) {
            SC_THREAD(write_main);
        }
    
    void write_main() {
        const char *str = "Visit www.systemc.org!\n";
        const char *p = str;
        while(true) {
            if(rand() & 1) {
                out->write(*p++);
                if(!*p) p = str;
            }
            wait(1, SC_NS);
        }
    }
};

class consumer : public sc_module {
    public:
        sc_port<sc_fifo_in_if<char>> in;
        SC_HAS_PROCESS(consumer);
        consumer(sc_module_name name) : sc_module(name) {
            SC_THREAD(read_main);
        }
    
    void read_main() {
        char c;
        while(true) {
            if(rand() & 1) {
                in->read(c);
                cout << c;
            }
            wait(1, SC_NS);
        }
    }
};

class top : public sc_module {
    public:
        sc_fifo<char> fifo_inst;
        producer prod_inst;
        consumer cons_inst;
        top(sc_module_name name, int size) : sc_module(name),
            fifo_inst("Fifo1", size),
            prod_inst("Producer1"),
            cons_inst("Consumer1"),
            { prod_inst.out(fifo_inst);
              cons_inst.in(fifo_inst);
            }
        
        int sc_main(int argc, char *argv[]) {
            int size = 10;
            top top1("Top1", size);
            sc_start(1000, SC_NS);
            cout << endl << endl;
            return 0;
        }
    
};
```
**RTL Implementation of FIFO**

- **Definition of input/output**

  - **Write Data**
  - **Read Data**

  - **Request to Write**
  - **Request to Read**

  - **Signal if there are free spaces in the FIFO**
  - **Signal if there are data stored in the FIFO**

  - **Primitive Channels** (sc_signal)

  - **hw_fifo**

- **Signals**:
  - `data_in`
  - `valid_in`
  - `ready_out`
  - `data_out`
  - `valid_out`
  - `ready_in`
  - `clock`
template<class T> class hw_fifo: public sc_module
{
public:
    sc_in<bool> clk;
    sc_in<T> data_in;
    sc_in<bool> valid_in;
    sc_out<bool> ready_out;
    sc_out<T> data_out;
    sc_out<bool> valid_out;
    sc_in<bool> ready_in;

    SC_HAS_PROCESS(hw_fifo);
    hw_fifo(sc_module_name name, unsigned size):
        sc_module(name, _size(size))
    {
        assert (size > 0);
        _first = _items = 0;
        _data = new T[_size];
    }

    SC_METHOD(hw_fifo_method);
    sensitive << clk.pos();

    ready_out.initialize(true);
    valid_out.initialize(false);
};

~hw_fifo() {delete[] _data;}

protected:

    void hw_fifo_method()
    {
        if (valid_in.read() && ready_out.read())
        {
            // store new data item into fifo
            _data[(_first + _items) % _size] = data_in;
            ++_items;
        }

        if (ready_in.read() && valid_out.read())
        {
            // discard data item that was just read from fifo
            --_items;
            _first = (_first + 1) % _size;
        }

        // Update all output signals.
        // Valid on next delta-cycle
        ready_out = (_items < _size);
        valid_out = (_items > 0);
        data_out = _data[_first];
    }
template <class T> class hw_fifo_wrapper
    : public sc_module, public sc_fifo_in_if <T>,
      public sc_fifo_out_if <T>
{
    public:
        sc_in<bool>   clk;
    Protected:
        // embedded channels
        sc_signal<T>      write_data;
        sc_signal<bool>   write_valid;
        sc_signal<bool>  write_ready;
        sc_signal<T>     read_data;
        sc_signal<bool> read_valid;
        sc_signal<bool> read_ready;

        // embedded module
        hw_fifo<T> hw_fifo;
    public: hw_fifo_wrapper(sc_module_name name, unsigned size)
        : sc_module(name), hw_fifo("hw_fifo1", size)
    {
            hw_fifo.clk(clk);
            hw_fifo.data_in (write_data);
            hw_fifo.valid_in (write_valid);
            hw_fifo.ready_out(write_ready);
            hw_fifo.data_out (read_data);
            hw_fifo.valid_out (read_valid);
            hw_fifo.ready_in (read_ready);
    }

    virtual void write(const T& data)
    {
            write_data = data; // Valid on next delta-cycle
            write_valid = true; // Valid on next delta-cycle
            do {
                    wait(clk->posedge_event());
                    while (write_ready != true);
                    write_valid = false;
            } while (write_valid != false);
    }

    virtual T read()
    {
            read_ready = true; // Valid on next delta-cycle
            do {
                    wait(clk->posedge_event());
                    while (read_valid != true);
            } while (read_valid != true);
            return read_data.read();
    }

    virtual void read(T& d) {d = read();} // Provide dummy implementations for unneeded
    // sc_fifo<T> interface methods;
    virtual bool nb_read(T& d) { assert(0); return false; } // sc_fifo<T> interface methods;
    virtual bool nb_write(const T& d) { assert(0); return false; } // sc_fifo<T> interface methods;
    virtual int num_available() const
    {
            assert(0); return 0;
    }
    virtual int num_free() const
    {
            assert(0); return 0;
    }
};
class top : public sc_module
{
  public:
  hw_fifo_wrapper<char> fifo_inst;
  producer prod_inst;
  consumer cons_inst;
  sc_clock clk;

  top(sc_module_name name, int size)
  : sc_module(name),
    fifo_inst("Fifo1", size),
    prod_inst("Producer1"),
    cons_inst("Consumer1"),
    clk("c1", 1, SC_NS)
  {
    prod_inst.out(fifo_inst);
    cons_inst.in(fifo_inst);
    fifo_inst.clk(clk);
  }
};

int sc_main(int argc, char *argv[])
{
  int size = 10;
  top top1("Top1", size);
  sc_start(1000, SC_NS);
  cout << endl <<endl;
  return 0;
}