EEL 5722C
Field-Programmable Gate Array Design

Lecture 16: System-Level Modeling in SystemC 2.0

Prof. Mingjie Lin

* Stuart Swan, “An Introduction to System-Level Modeling in SystemC 2.0,” Cadence Design Systems, 2001.

Debugging

Text-based Debugging

- C++ “printf” debugging

```
printf(“Hello World”);

cout << “Hello World” << endl;
```

Text-based Debugging

- Constructor Debugging
  - Find out how your design is built up when the simulation starts.
  - Use the **name()** method to identify SystemC classes:

```
SC_CTOR(nand2) {
  cout << "Constructing nand2 " << name() << endl;
  ...
  ...
}

OUTPUT:
Constructing stim
Constructing nand2 exor2.N1
Constructung nand2 exor2.N2
```

Text-based Debugging

- Debugging methods available on all SystemC objects:
  - const char* name()
    - Returns the name of the object
  - const char* kind()
    - Returns the object's sub-class name
  - void print(ostream& out)
    - Prints the object's name to the output stream
  - void dump(ostream& out)
    - Prints the objects diagnostic data to the output stream.

Text-based Debugging

- Debugging threads and methods
  - All SystemC data types can be “printed” to **cout**.
    - e.g.: print inputs A, B, and F to **cout** in a table:

```
OUTPUT:

Time      A B F
10 ns     1 0 0
20 ns     1 1 0
30 ns     1 1 1
40 ns     0 0 1
```

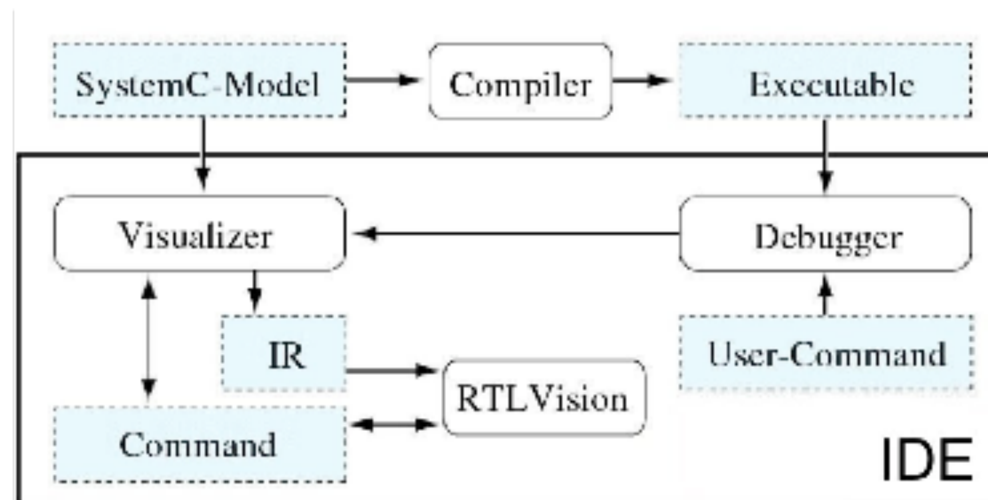
Text-based Debugging

```
SC_MODULE(mon)
{
  sc_in<bool> A,B,F;
  sc_in<bool> Clk;

  void monitor()
  {
    cout << "Time   A   B   F" << endl;
    while (true)
    {
      cout <<  sc_time_stamp() << ", ";
      cout <<  A.read() << ", ";
      cout <<  B.read() << ", ";
      cout <<  F.read() << endl;
      wait();    // wait for 1 clock cycle
    }
  }

  SC_CTOR(mon)
  {
    SC_THREAD(monitor);
    sensitive << Clk.pos();
  }
}
```

Advanced Debugging

- Standard C++ debugging tools
  - GDB, etc...

- SystemC-specific debuggers and visualizers.



Advanced Debugging

Schema View

Info Box

Cone View

Label

Wave-form Debugging

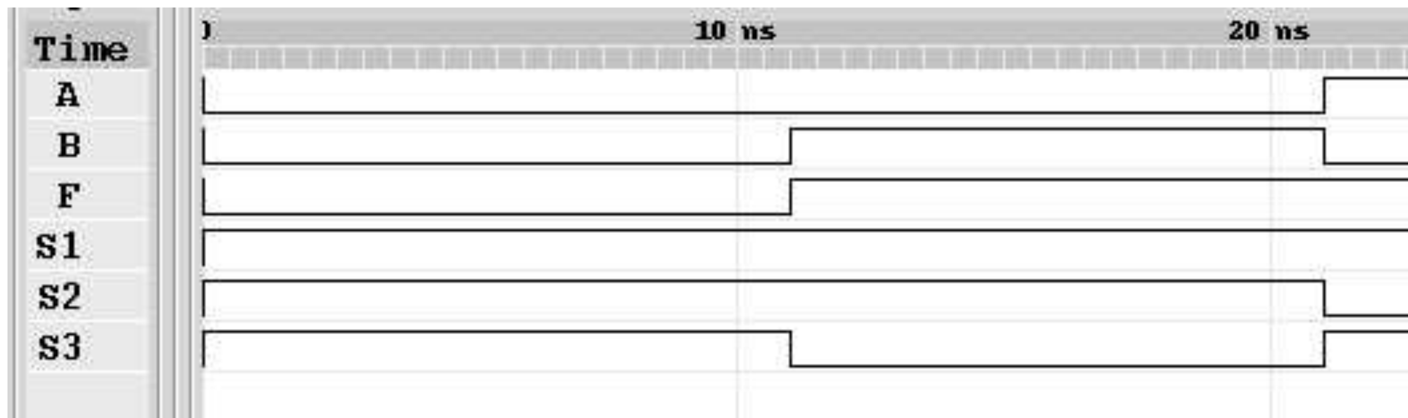
- Requires adding additional SystemC statements to **sc_main()**
  - Wave-form data written to file as simulation runs.
  - Sequence of operations:
    - Declare and create the trace file
    - Register signals or events for tracing
    - Run the simulation
    - Close the trace file

Wave-form Tracing

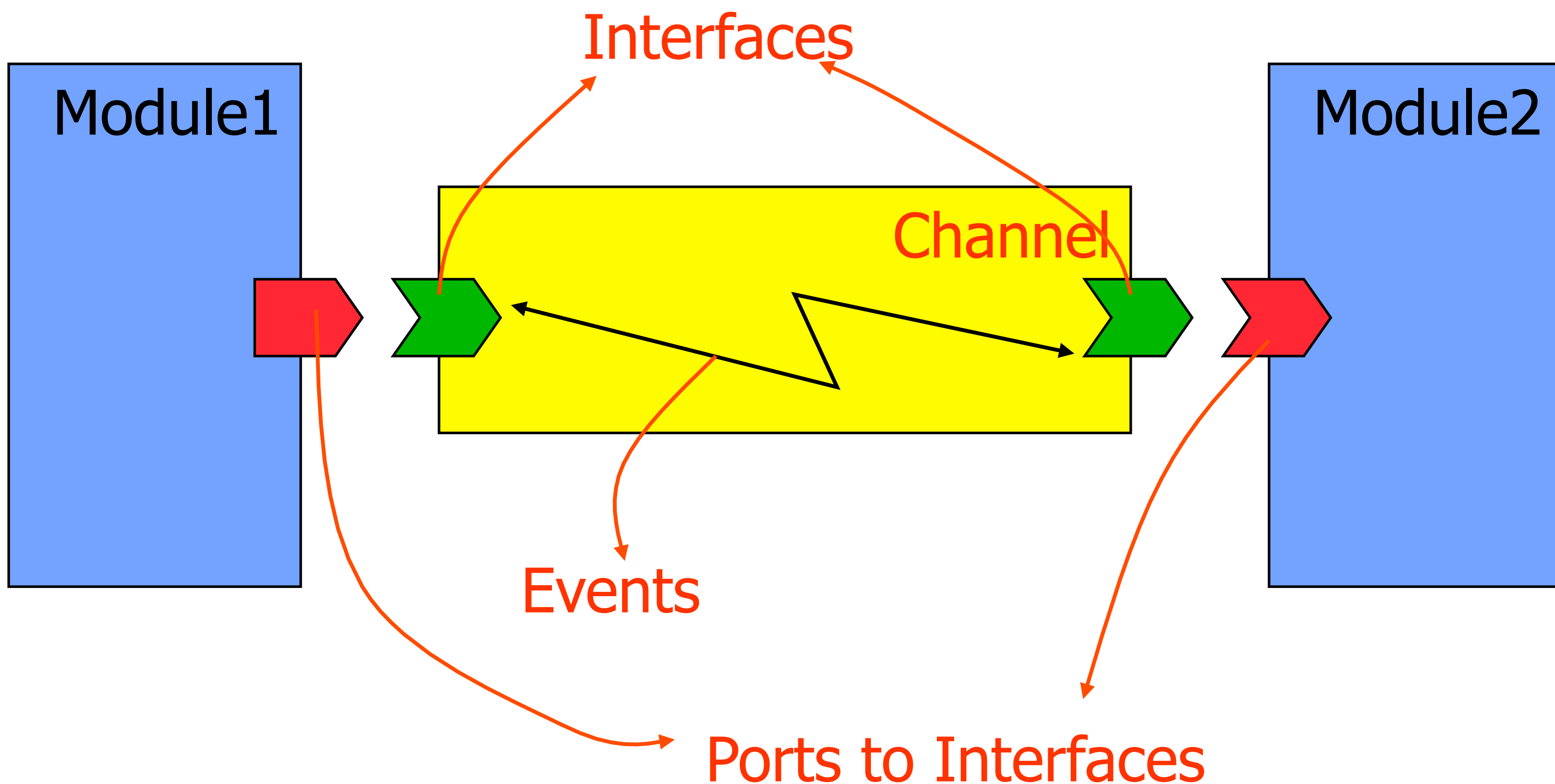
```
int sc_main(int argc, char* argv[])
{
  sc_signal<bool> ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS,0.5, 1, SC_NS);

  // Set up simulation
  ...

  // Set up trace file
  sc_trace_file* Tf;
  Tf = sc_create_vcd_trace_file("traces");          // Create Trace File
  ((vcd_trace_file*)Tf)->sc_set_vcd_time_unit(-9);   // Set time unit
  sc_trace(Tf, ASig  , "A" );                         // Register signals
  sc_trace(Tf, BSig  , "B" );                         // and variables.
  sc_trace(Tf, FSig  , "F" );
  sc_trace(Tf, DUT.S1, "S1");
  sc_trace(Tf, DUT.S2, "S2");
  sc_trace(Tf, DUT.S3, "S3");

  sc_start();  // run forever                         // Start the simulation
  sc_close_vcd_trace_file(Tf);                        // Close the trace file
  return 0;
}
```

Wave-form Tracing

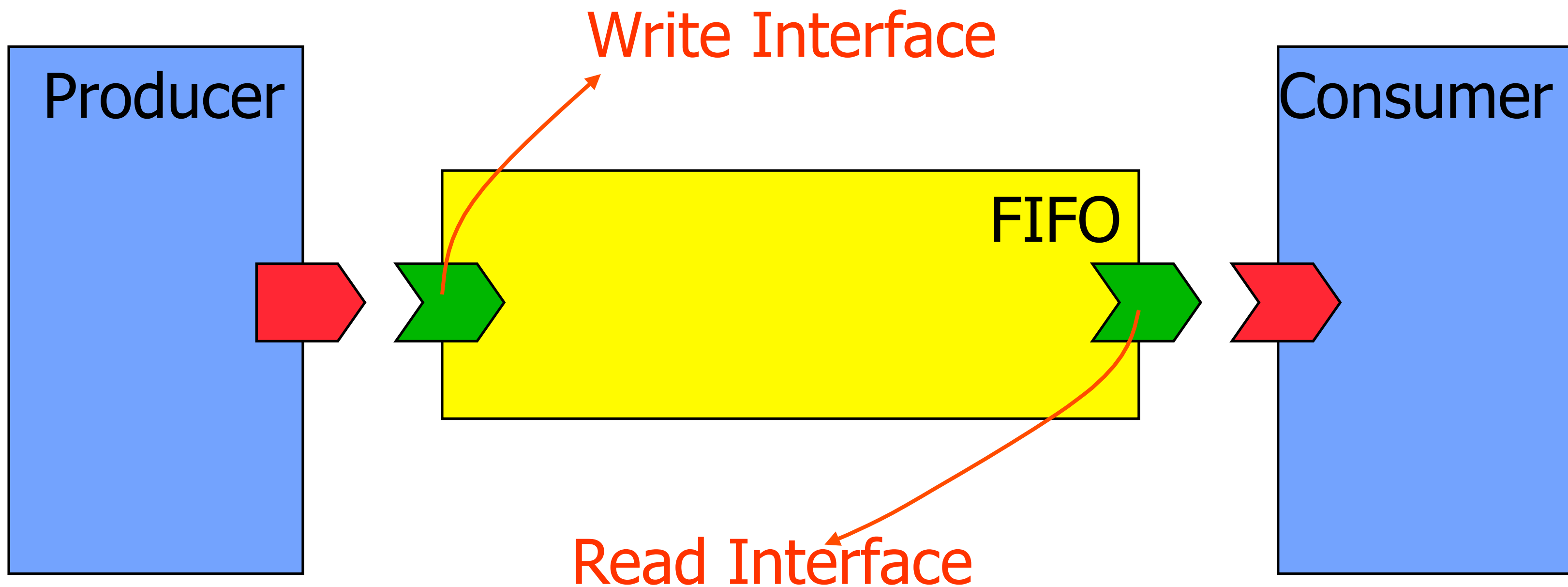
- Sample Output



Communication and Synchronization



A Communication Modeling Example: FIFO



Problem definition: FIFO communication channel with blocking read and write operations

Source available in SystemC installation, under "`examples\systemc`" subdirectory

FIFO Example: Declaration of Interfaces

```
class write_if : public sc_interface
{
   public:
        virtual void write(char) = 0;
        virtual void reset() = 0;
};

class read_if : public sc_interface
{
   public:
        virtual void read(char&) = 0;
        virtual int num_available() = 0;
};
```

FIFO Example:
Declaration of *FIFO channel*

```
class fifo: public sc_channel,
   public write_if,
   public read_if
{
   private:
        enum e {max_elements=10};
        char data[max_elements];
        int  num_elements, first;
        sc_event write_event,
                 read_event;
        bool fifo_empty() {…};
        bool fifo_full() {…};

   public:
        SC_CTOR(fifo) {
           num_elements = first=0;
        }
```

```
void write(char c) {
   if ( fifo_full() )
        wait(read_event);
   data[ <you say> ]=c;
   ++num_elements;
   write_event.notify();
}

void read(char &c) {
   if( fifo_empty() )
        wait(write_event);
   c = data[first];
   --num_elements;
   first = <you say>;
   read_event.notify();
}
```

Declaration of *FIFO channel* (cont'd)

```
   void reset() {
        num_elements = first = 0;
   }

   int num_available() {
        return num_elements;
   }
}; // end of class declarations
```

FIFO Example (cont'd)

- All channels must
  - be derived from `sc_channel` class
    - SystemC internals (kernel\sc_module.h)

      `typedef sc_module sc_channel;`
  - be derived from one (or more) classes derived from `sc_interface`
  - provide implementations for all pure virtual functions defined in its parent *interfaces*

FIFO Example (cont'd)

- Note the following extensions beyond SystemC 1.0
  - `wait()` call with arguments => dynamic sensitivity
    - `wait(sc_event)`
    - `wait(time) // e.g. wait(200, SC_NS);`
    - `wait(time_out, sc_event) //wait(2, SC_PS, e);`
  - Events
    - are the fundamental synch. primitive in SystemC 2.0
    - Unlike signals,
      - have no type and no value
      - always cause sensitive processes to be resumed
      - can be specified to occur:
        - immediately/ one delta-step later/ some specific time later

The `wait()` function

```
// wait for 200 ns.
sc_time t(200, SC_NS);
wait( t );

// wait on event e1, timeout after 200 ns.
wait( t, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( t, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( t, e1 & e2 & e3 );

// wait for 200 clock cycles, SC_CTHREAD only (SystemC 1.0).
wait( 200 );

// wait one delta cycle.
wait( 0, SC_NS );

// wait one delta cycle.
wait( SC_ZERO_TIME );
```

The `notify()` method of `sc_event`

- Possible calls to notify():

```
sc_event my_event;

my_event.notify(); // notify immediately

my_event.notify( SC_ZERO_TIME ); // notify next delta cycle

my_event.notify( 10, SC_NS ); // notify in 10 ns

sc_time t( 10, SC_NS );
my_event.notify( t ); // same
```

Completing the Comm. Modeling Example

```
SC_MODULE(producer) {
   public:
        sc_port<write_if> out;

   SC_CTOR(producer) {
        SC_THREAD(main);
   }

   void main() {
        char c;
        while (true) {
           out->write(c);
           if(…)
                out->reset();
        }
   }
};
```

```
SC_MODULE(consumer) {
   public:
        sc_port<read_if> in;

   SC_CTOR(consumer) {
        SC_THREAD(main);
   }

   void main() {
        char c;
        while (true) {
           in->read(c);
           cout<<
             in->num_available();
        }
   }
};
```

Completing the Comm. Modeling Example (cont’d)

```
SC_MODULE(top) {
   public:
        fifo *afifo;
        producer *pproducer;
        consumer *pconsumer;

   SC_CTOR(top) {
        afifo = new fifo(“Fifo”);

        pproducer=new producer(“Producer”);
        pproducer->out(afifo);

        pconsumer=new consumer(“Consumer”);
        pconsumer->in(afifo);
   };
```

Completing the Comm. Modeling Example (cont'd)

- Note:
  - Producer module
    - sc_port<*write_if*> out;
      - Producer can only call member functions of *write_if* interface
  - Consumer module
    - sc_port<*read_if*> in;
      - Consumer can only call member functions of *read_if* interface
      - e.g., Cannot call `reset()` method of write_if
  - Producer and consumer are
    - unaware of how the channel works
    - just aware of their respective *interfaces*
  - Channel implementation is hidden from communicating modules

Completing the Comm. Modeling Example (cont'd)

- Advantages of separating communication from functionality
  - Trying different communication modules
  - Refine the FIFO into a software implementation
    - Using queuing mechanisms of the underlying RTOS
  - Refine the FIFO into a hardware implementation
    - Channels can contain other channels and modules
      - Instantiate the hw FIFO module within FIFO channel
      - Implement read and write interface methods to properly work with the hw FIFO
      - Refine read and write interface methods by inlining them into producer and consumer codes

Final issues

- Come by my office hours (right after class)
- Any questions or concerns?