



System Level Modeling Language

Outline



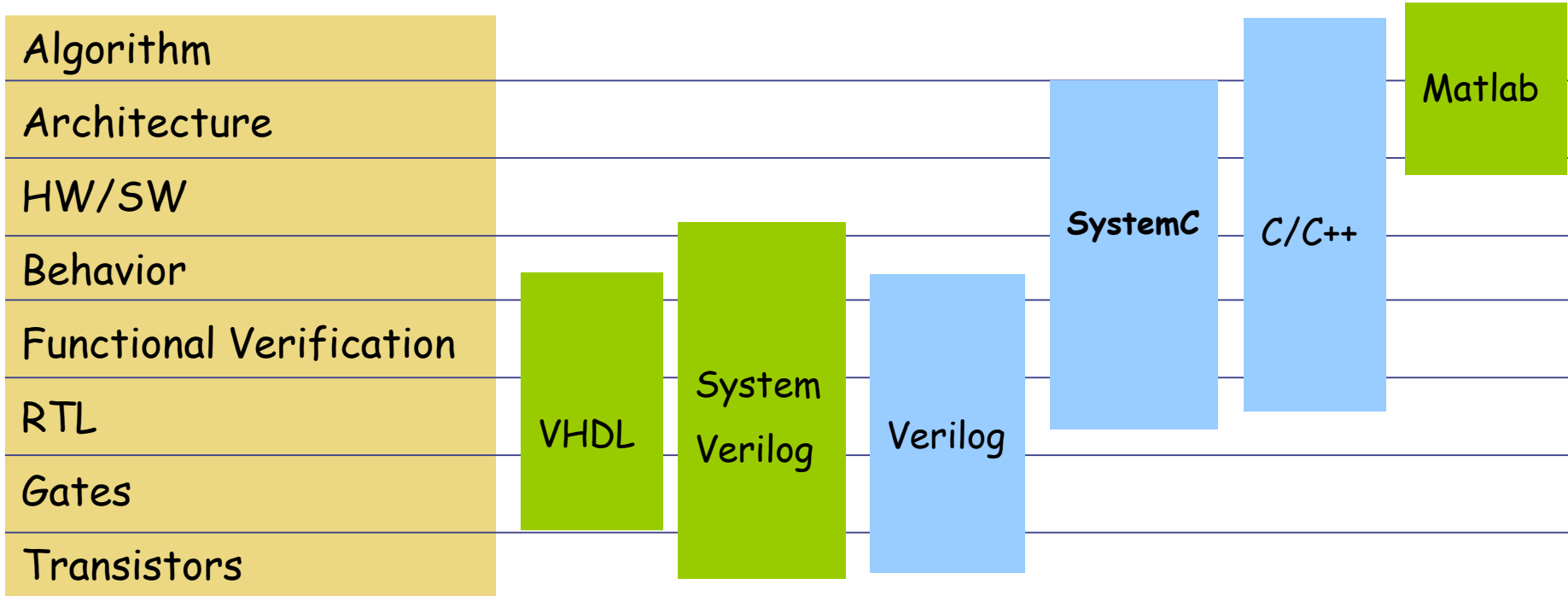
- *System Design Trend*
- *Introduction to SystemC*
- *Model Construction*
 - ◆ *Module*
 - ◆ *Port, interface and channel*
 - ◆ *Data type*
 - ◆ *Process*
 - ◆ *Clock Object*
- *Examples of SystemC*

The Need of SystemC (1/2)



- Model **HARDWARE** in high abstraction level
- Cover **multiple levels** of abstraction
- Render a **platform** to co-verify HW&SW
- Identify **potential problems** of SoC design, decreasing the risk of project failure

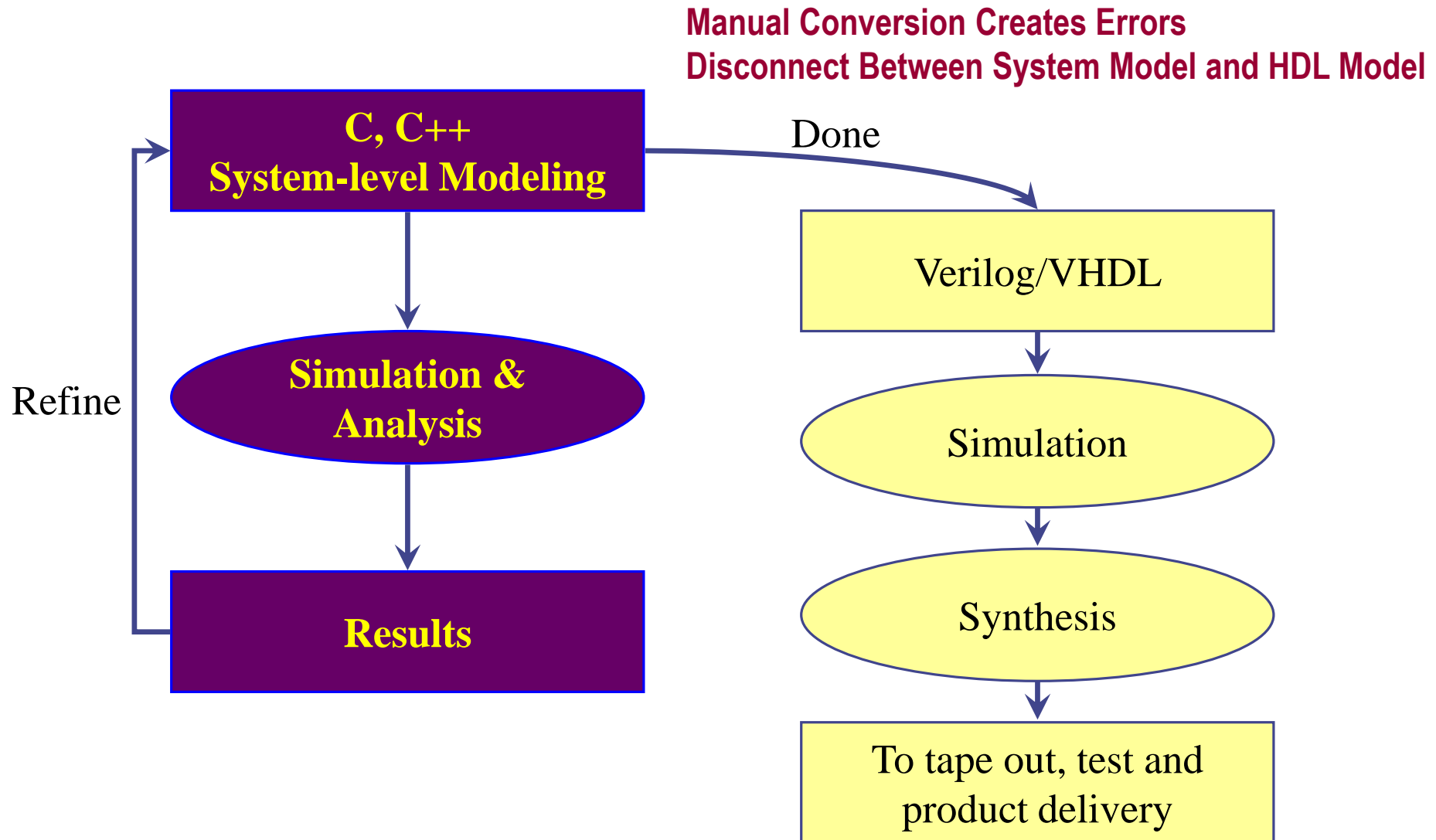
The Need of SystemC (2/2)



[1]

Introduction to SystemC

Conventional System Design Methodology



SystemC Overview (1/2)



- The dream to realize the unison of HW/SW designing languages
 - ◆ A unified design environment
- Version 1: it is just another HDL, not much to do with system-level designing
- Version 2: with the adding of channel, now it is a serious system-level language
 - ◆ channels, interfaces and ports
 - ◆ much more powerful modeling for Transaction Level
- ...
- Future SystemC 3.0
 - ◆ Modeling of OSs
 - ◆ Support of embedded SW models

SystemC Overview (2/2)



- Is a C++ class library and a methodology that one can use to effectively create cycle-accurate models of functions, hardware architecture, and interfaces of the SoC and system-level designs
- One can use SystemC and standard C++ development tools to
 - ◆ create a system-level model
 - ◆ quickly simulate to validate and optimize the design
 - ◆ explore various algorithms
 - ◆ provide the hardware and software development team with an executable specification of the system

SystemC Highlights



- **Modules**: component
- **Processes**: functions, SC_THREAD & SC_METHOD
- **Ports**: I/O ports
- **Signals**: wires
- Rich set of port and signal types
- Rich set of **data types**
- **Clocks**
- **Event-based (cycle-accurate) simulation**: ultra light-weight and fast
- **Multiple abstraction levels**
- **Communication protocols**: **channel & interface**
- Debugging support: runtime error checking
- Waveform tracing: VCD, WIF and ISDB formats

SystemC & C++



- SystemC is a set of C++ class definitions and a methodology for using these classes
- C++ class definition means **systemc.h** and the matching library
- Methodology means the use of simulation kernel and modeling
- You can use all of the C++ syntax, semantics, run time library, STL (Standard Template Library) and such
- However you need to follow SystemC methodology closely to make sure the simulation executes correctly

Note



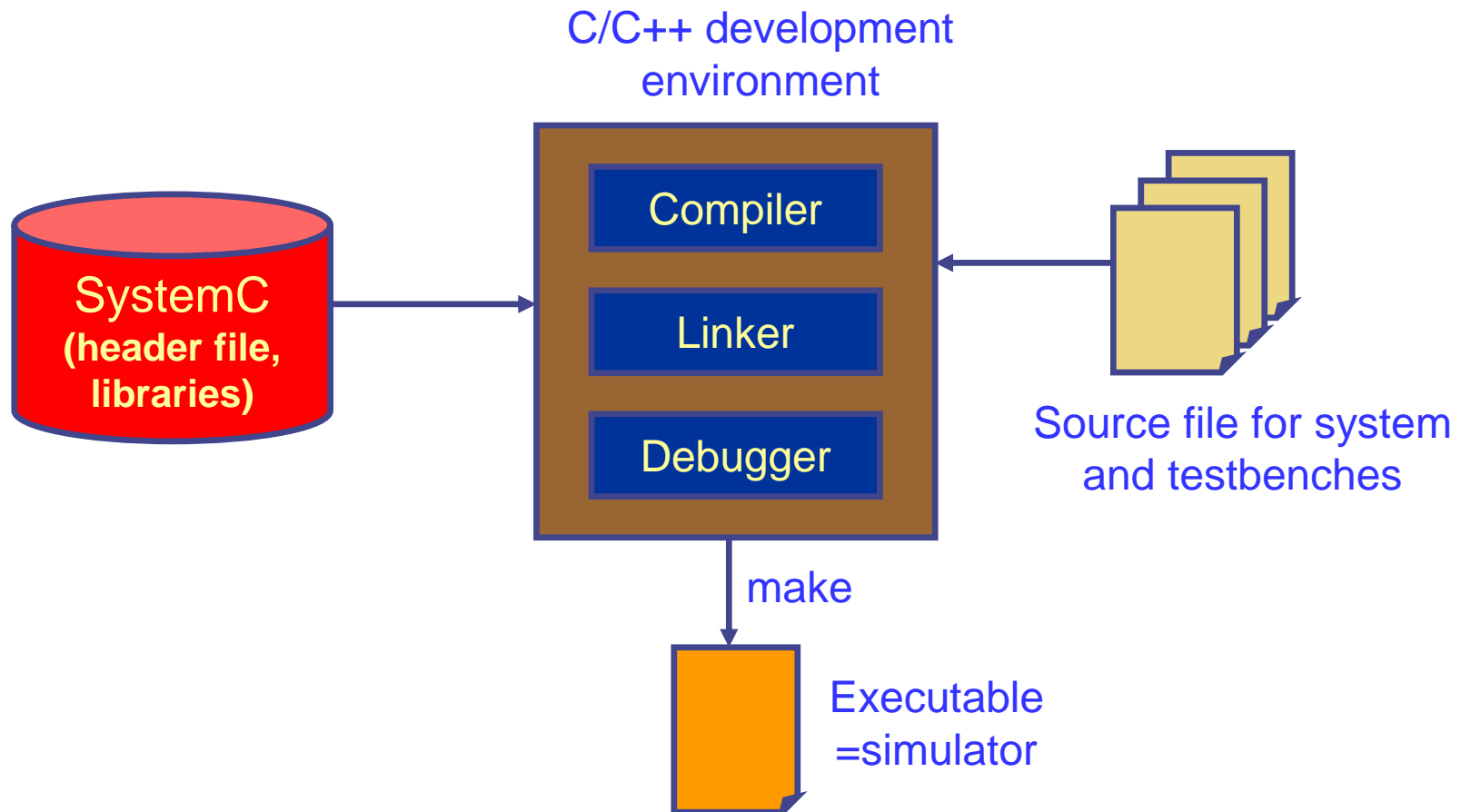
- SystemC is a **Hardware Description Language (HDL)** from **system-level** down to **gate level**
- SystemC **does not** model **software**
- SystemC **does not** run faster, higher abstraction level does

System-Level Language



- To be categorized as a system-level language, the **simulation SPEED** is the key
- The simulation speed should take **no 1,000 time slower than the real HW**
 - ◆ In another word, **1 second** of HW execution time equals **16 minutes and 40 seconds** simulation time
- To achieve this kind of performance, the system is best modeled in **transaction level**, e.g. **token based**

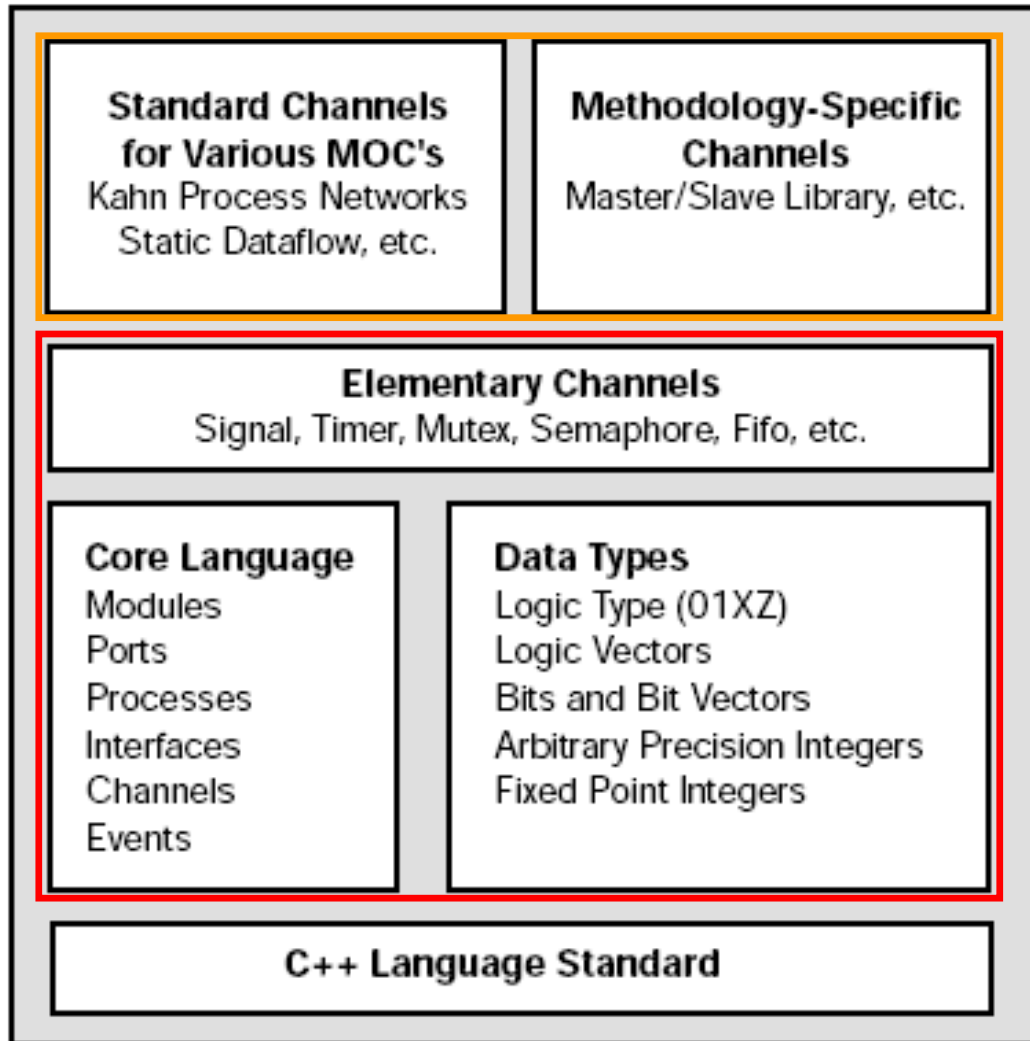
SystemC Design Flow



Language Architecture



MOC: model of computation



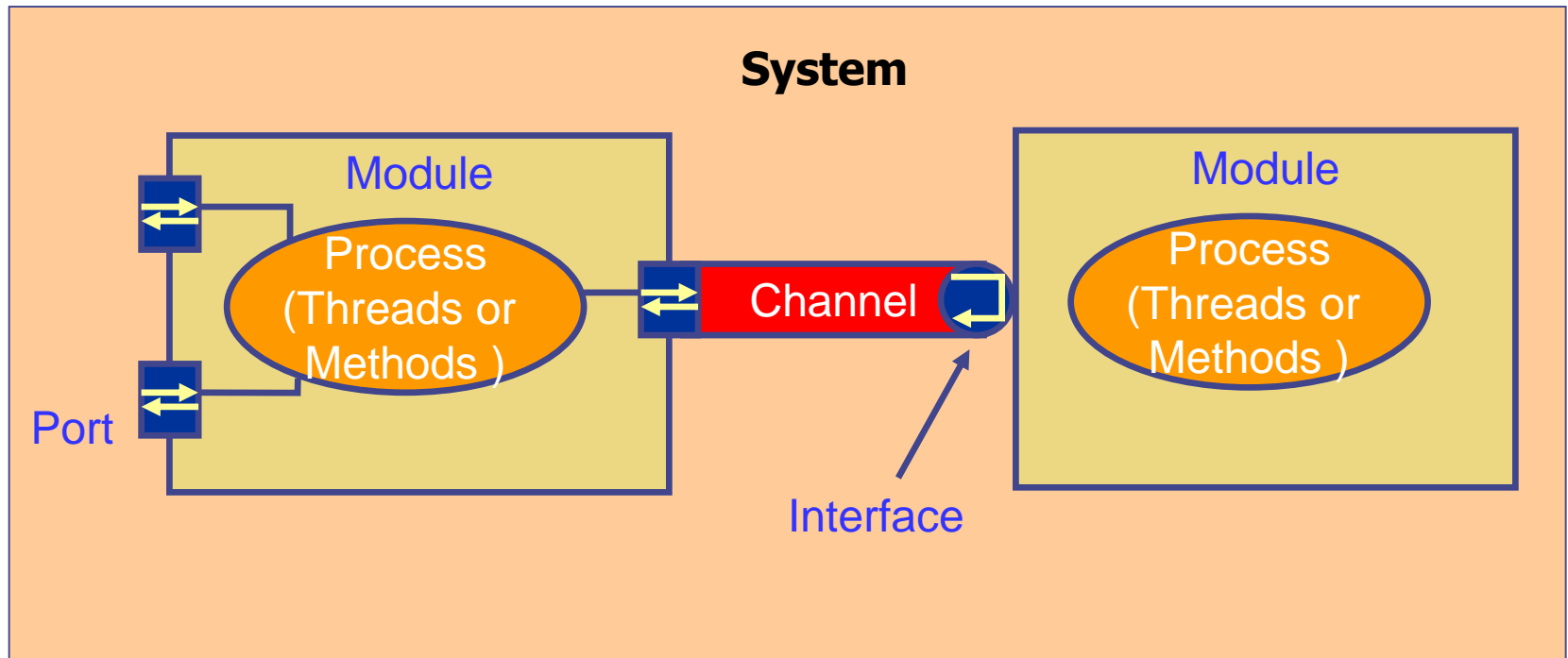
Not-standard

SystemC standard

Basic Structure

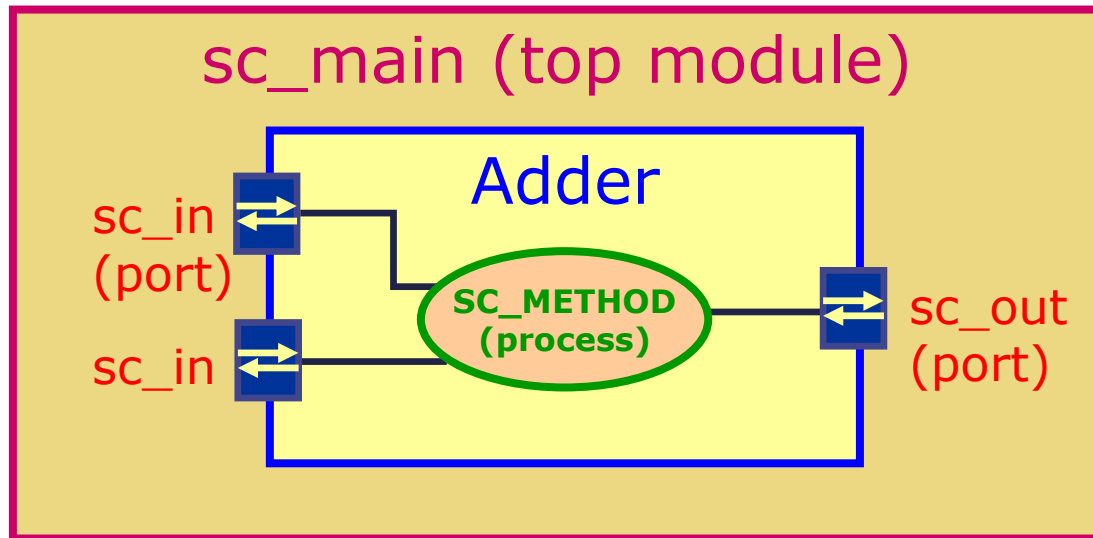
Basic Structure of SystemC model

- ◆ Module
- ◆ Port, interface and channel
- ◆ Process



Example of Adder

■ Behavioral model of adder



Source Code

■ SC_MODULE macro for declaration

```
#include <systemc.h>

SC_MODULE(Adder)
{
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> c;

    void compute();

    SC_CTOR(Adder)
    {
        SC_METHOD(compute);
        sensitive << a << b;
    }
};
```

Adder.h

```
#include "Adder.h"

void Adder::compute()
{
    c = a + b;
}
```

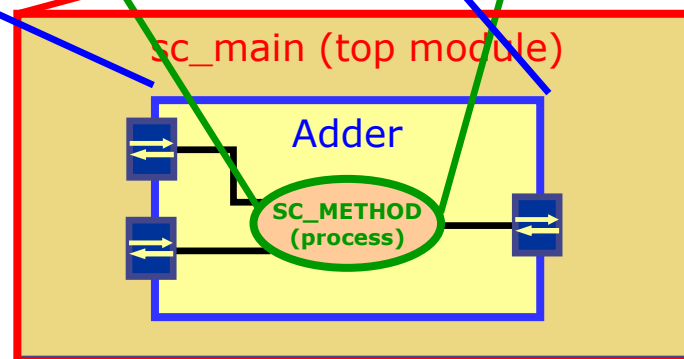
Adder.cpp

```
#include <systemc.h>
#include "Adder.h"

int sc_main(int argc, char* argv[])
{
    sc_signal<int> sig_a, sig_b, sig_c;
    Adder my_adder("my_adder");
    my_adder(sig_a, sig_b, sig_c);

    sc_start(1000, SC_SEC);
    return 0;
}
```

main.cpp



Model Construction

Construct Model



- Module
- Port, interface and channel
- Data type and Time
- Process

Modules

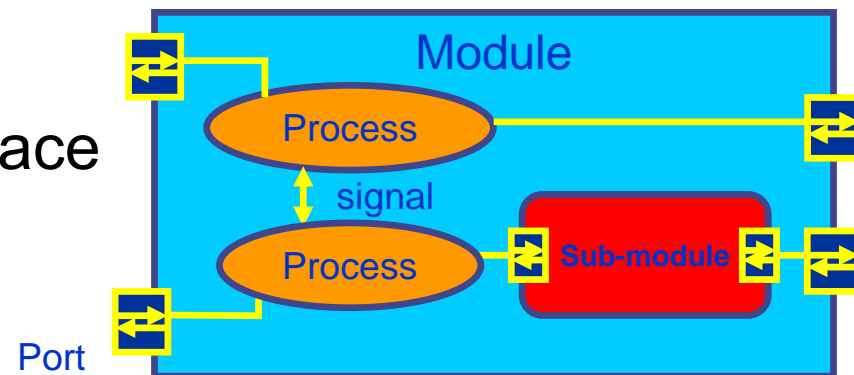
- A **module** is the **basic structural building block** in SystemC, the same as Verilog “module”
- Keyword: **SC_MODULE**
- Can contain
 - ◆ **Ports**: communicate with outside
 - ◆ **Process**: describe functionality of module
 - ◆ **Internal** data and channels (sc_signal, ...etc.)
 - ◆ **Other modules** (sub-module)
- Can access a channel's interface

```
#include <systemc.h>

SC_MODULE(Adder)
{
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> c;

    void compute();

    SC_CTOR(Adder)
    {
        SC_METHOD(compute);
        sensitive << a << b;
    }
};
```



Modules - Semantic

■ Module has **constructor**:

- ◆ To declare **process** and **sensitivity list** (if necessary)

```
SC_MODULE ( module_name ){  
    // ports, porcess, internal data  
    sc_in< data_type > input;  
    sc_inout < data_type > inout;  
    sc_out< data_type > out;  
  
    SC_CTOR( module_name ){  
        // constructor  
        // process declaration, sensitivities  
    }  
}
```

```
class module_name: public sc_module{  
public:  
    // ports, porcess, internal data  
    sc_in< data_type > input;  
    sc_inout < data_type > inout;  
    sc_out< data_type > out;  
    sc_module( sc_module_name ){  
        // constructor  
        // process declaration, sensitivities  
    }  
}
```

```
#include <systemc.h>  
  
SC_MODULE(Adder)  
{  
    sc_in<int> a;  
    sc_in<int> b;  
    sc_out<int> c;  
  
    void compute();  
  
    SC_CTOR(Adder)  
    {  
        SC_METHOD(compute);  
        sensitive << a << b;  
    }  
};
```

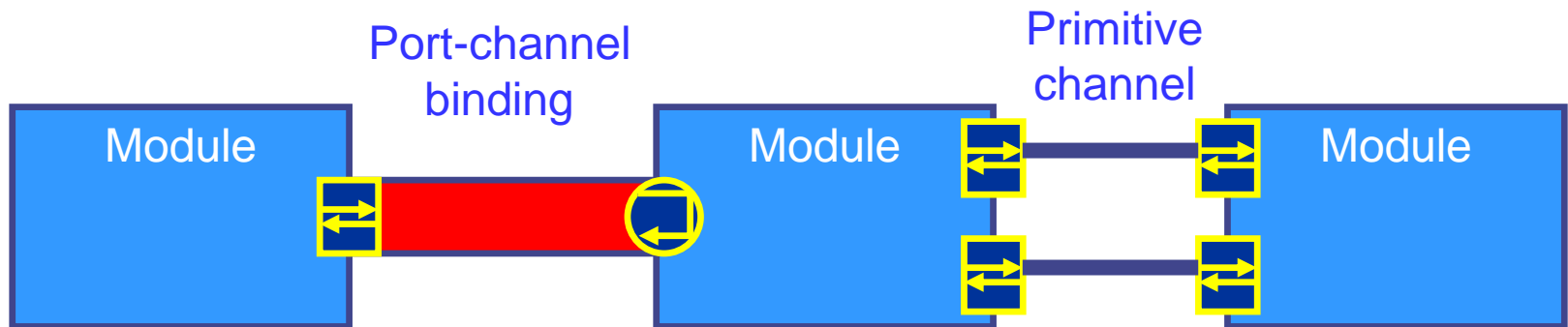
Construct Model



- Module
- Port, interface and channel
- Data type and Time
- Process

Port, Interface and Channel

- Flexibility and variable level abstraction
- They are like
 - ◆ **Channel**: the workhorse for holding and transmitting data
 - ◆ **Interface**: a “window” in a channel that describes the set of operation
 - ◆ **Port**: the proxy object that facilitates access to channel through interface
- They can be bound if the channel or port has compatible interface in compile-time and run-time

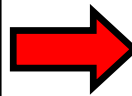


Ports

- Used by the module to **connect** to other modules
- Connect to **channels** through **interface**
- Three modes of operation: (inherit from class **sc_port**)
 - ◆ Input: **sc_in**<T> **sc_port**<**sc_in_if**<T> >
 - ◆ Output: **sc_out**<T> **sc_port**<**sc_out_if**<T> >
 - ◆ Inout: **sc_inout**<T> **sc_port**<**sc_inout_if**<T> >

```
Verilog

module module_name (module_port);
{
    // ports
    input< range > input;
    inout < range > inout;
    output< range > out;
}
}
```

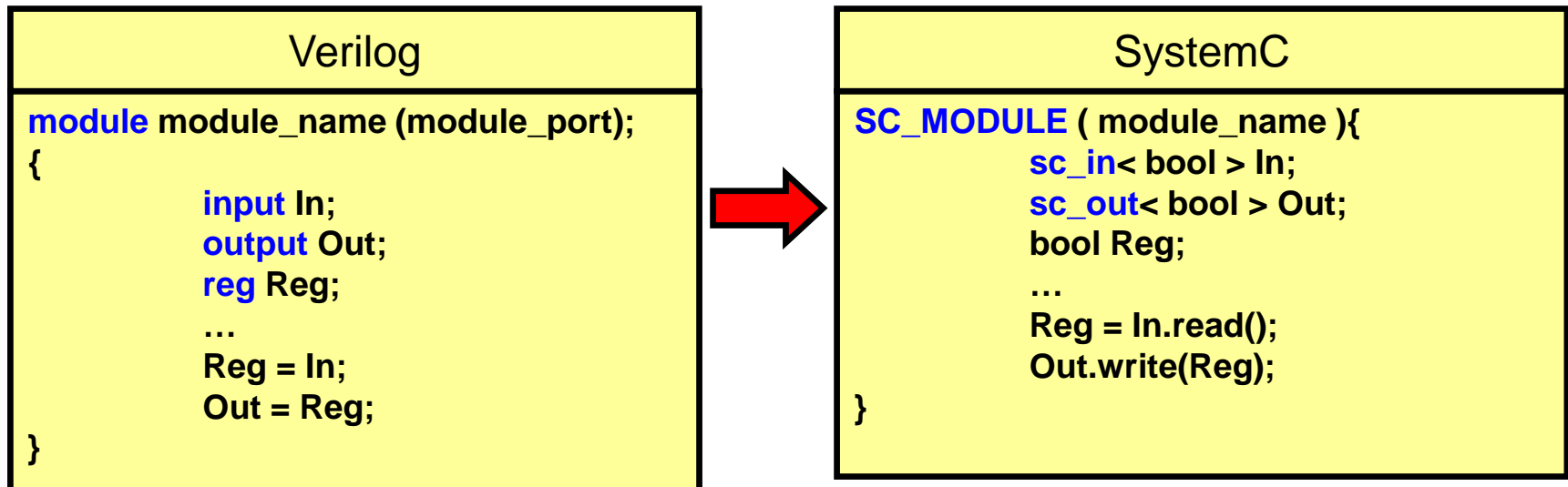


```
SystemC

SC_MODULE ( module_name ){
    // ports
    sc_in< data_type > input;
    sc_inout < data_type > inout;
    sc_out< data_type > out;
}
}
```


Access Ports

- sc_in, sc_out, sc_inout have default **channel method**:
 - ◆ read(), write()

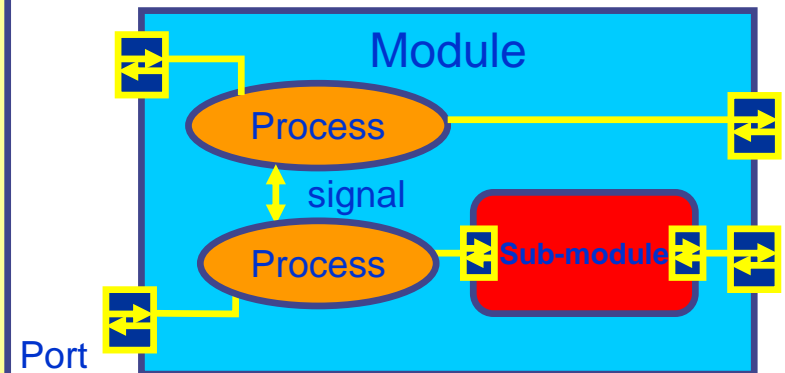


Port-Channel Access

■ For IP reuse and tool support, SystemC 2.0 defines design style:

- ◆ **Inter-module** level communication
 - ▶ Use **port** to connect module to channel P.25, P.94, P.97, P.100
- ◆ **Intra-module** level communication
 - ▶ Access channel's interface directly ("**port-less**") P.32, P.34

```
SC_MODULE ( mod1 ){  
    sc_in< int > in;  
    sc_out< int > out;  
  
    sc_mutex mutex; // channel  
  
    ...  
    mutex.lock(); //direct access to channel  
    // critical section code  
    ...  
    mutex.unlock(); //direct access to channel  
    ...  
}
```



Interfaces

- Define **methods** that **channels must implement**
- All interfaces must be derived from base class **sc_interface**
 - ◆ Ports connect to channels through interfaces
 - ◆ Ports **only** can see the **channels' method** defined in **interface** which connect to

```
template <class T>
class sc_read_if: virtual public sc_interface
{
public:
    // interface method
    virtual const T& read() const = 0;
};
```

```
template <class T>
class sc_write_if: virtual public sc_interface
{
public:
    // interface method
    virtual void write(const T&) = 0;
};
```

Interface Derivation

- Define **read/write** interface by deriving from **read** interface and **write** interface

```
template <class T>
class sc_read_write_if: public sc_read_if<T>, public sc_write_if<T>
{
};
```

- Interfaces for **sc_fifo channel**
 - ◆ **sc_fifo_in_if**: read(), nb_read(), ... methods
 - ▶ **read()**: if not empty, read; if empty, **wait** until available and then read
 - ▶ **nb_read()**: returns false, when fifo is empty (**does not wait**); otherwise reads and returns true
 - ◆ **sc_fifo_out_if**: write(), nb_write(), ... methods

Channels



■ Channels:

- ◆ Implement one or more interfaces, serve for communication functionality
- ◆ Use to hold and transmit data
- ◆ Can connect multiple modules
- ◆ SystemC 2.0 allow user to create their own channel types

■ Classify:

- ◆ Primitive channels
- ◆ Hierarchical channels

Channels Classify



- **Primitive** channels (derived from `sc_prim_channel`)
 - ◆ Atomic, doesn't contain other SystemC structures
 - ◆ Not exhibit structure
 - ◆ Lack internal processes
 - ◆ Can't access (directly) other primitive channels
- **Hierarchical** channels (derived from `sc_channel`)
 - ◆ Construct by module
 - ◆ Can have structure
 - ◆ Can contain other modules and processes
 - ◆ Can access (directly) other channels by using operator, ex: `fifo1.read()` or `fifo1.write()`

Primitive Channels



- Derived from base class `sc_prim_channel`
- Elementary channels
 - ◆ Hardware signal
 - ▶ `sc_signal<T>`
 - ◆ FIFO channel
 - ▶ `sc_fifo<T>`
 - ◆ Mutual-exclusion lock
 - ▶ `sc_mutex`

Hardware Signal Channel



- **sc_signal**<T> implement the interface
sc_signal_inout_if<T>

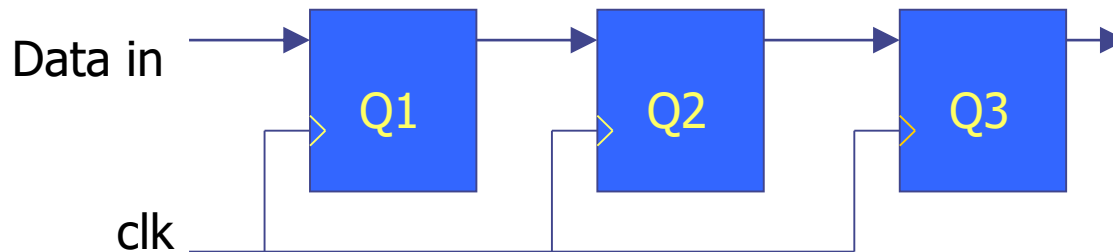
P.24

```
SC_MODULE ( controller ){
    sc_in< sc_logic > clk;
    sc_in< sc_logic > status;
    sc_out< sc_logic > count;
    sc_signal< sc_logic > lstat;
    state_machine *s1;

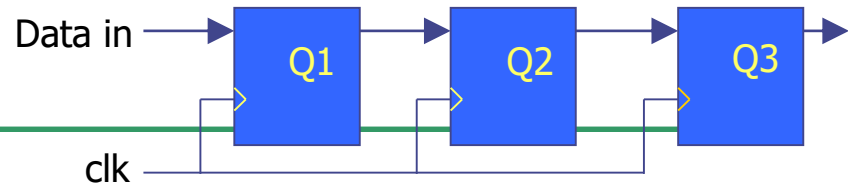
    SC_CTOR( controller ){
        s1 = new state_machine("s1"); // instantiate a module
        s1->clock(clk);
        s1->en(lstat); // port bound to signal
        s1->st(status);
    }
}
```


SC_signal

- For sequential logic, the **registers** is synchronized by clock
 - ◆ Those registers need to be updated with a **non-blocking** manner
- **SC_signal** meets this requirement



Example: Shifter



```
#include <systemc.h>
#include <iostream.h>
#include <stdlib.h>
SC_MODULE(shift_reg)
{
```

```
    sc_in_clk iclk;
    sc_signal<int> data_in;
    sc_signal<int> Q1;
    sc_signal<int> Q2;
    sc_signal<int> Q3;
```

```
    SC_CTOR(shift_reg)
    {
```

```
        SC_CTHREAD(sync_shift_reg, iclk.pos());
        data_in=0;
        Q1=0;
        Q2=0;
        Q3=0;
    }
```

```
    void sync_shift_reg()
    {
```

```
        for(;;)
        {
```

```
            data_in=rand()%100;
            Q1=data_in;
            Q2=Q1;
            Q3=Q2;
            wait(SC_ZERO_TIME);
```

```
            cout<<"At time "<<sc_time_stamp()<<" Q1: "<<Q1<<" Q2: "<<Q2<<" Q3:" <<Q3<<endl;
```

```
        }
```

```
    }
```

```
};
```

```
int sc_main(int argc, char *argv[])
{
    const sc_time t_PERIOD(20,SC_NS);

    sc_clock clk("clk",t_PERIOD);
    shift_reg ishift_reg("ishift_reg");
    ishift_reg.iclk(clk);

    sc_start(200,SC_NS);

    return 0;
}
```

At time	20 ns	Q1: 0	Q2: 0	Q3: 0
At time	40 ns	Q1: 38	Q2: 0	Q3: 0
At time	60 ns	Q1: 58	Q2: 38	Q3: 0
At time	80 ns	Q1: 13	Q2: 58	Q3: 38
At time	100 ns	Q1: 15	Q2: 13	Q3: 58
At time	120 ns	Q1: 51	Q2: 15	Q3: 13
At time	140 ns	Q1: 27	Q2: 51	Q3: 15
At time	160 ns	Q1: 10	Q2: 27	Q3: 51
At time	180 ns	Q1: 19	Q2: 10	Q3: 27

■ FIFO channel

- ◆ Provide blocking and nonblocking version
- ◆ `sc_fifo<T>` implements the interface `sc_fifo_in_if<T>` and `sc_fifo_out_if<T>`

■ Interfaces for `sc_fifo` channel

- ◆ `sc_fifo_in_if`: `read()`, `nb_read()`, ... methods
 - ▶ `read()`: if not empty, read; if empty, **wait** until available and then read
 - ▶ `nb_read()`: returns false, when fifo is empty (**does not wait**); otherwise reads and returns true
- ◆ `sc_fifo_out_if`: `write()`, `nb_write()`, ... methods

Mutual-exclusion Lock



■ Mutual-exclusion lock (`sc_mutex`)

- ◆ Model the behavior of a **mutual exclusion lock** as used to control access to a **resource shared by concurrent processes**
- ◆ A mutex will be in one of two exclusive states: **unlocked or locked**
- ◆ Only one process can **lock** a given mutex at one time
- ◆ A mutex can only be **unlocked** by the particular process that locked the mutex, but may be locked subsequently by a different process

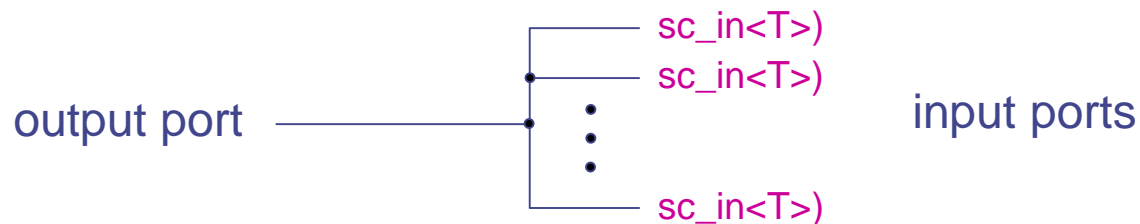
■ Member functions

- ◆ **lock()**: lock the mutex (wait until unlocked if in use)
- ◆ **trylock()**: non-blocking, return true if success, else false
- ◆ **unlock()**: free previously locked mutex

Channel Design Rule



sc_signal<T>	At most one output (sc_out<T>) or bi-directional port (sc_inout<T>) can connect
	Arbitrary number of input port (sc_in<T>) can connect
sc_fifo<T>	At most one input port (sc_in<T>) can connect
	At most one output port (sc_out<T>) can connect
	No bi-directional ports



Hierarchical Channels



- Use to model SoC communication infrastructure efficiently
- Ex: model On Chip Bus (OCB)

Construct Model



- Module
- Port, interface and channel
- Data type and Time
- Process

Data type

■ SystemC allow to use C++ data type and SystemC data type

- ◆ sc_bit – 2 value signal bit
- ◆ sc_logic – 4 value signal bit
- ◆ sc_int – 1 to 64 bit signed integer
- ◆ sc_uint – 1 to 64 bit unsigned integer
- ◆ sc_bigint – arbitrary sized signed integer
- ◆ sc_biguint – arbitrary sized unsigned integer
- ◆ sc_bv – arbitrary sized 2 value vector
- ◆ sc_lv – arbitrary sized 4 value vector
- ◆ sc_fixed – templated signed fixed point
- ◆ sc_ufixed – templated unsigned fixed point
- ◆ sc_fix – untemplated signed fixed point
- ◆ sc_ufix – untemplated unsigned fixed point

Type *sc_bit*

■ Two-valued '0' = false, '1' = true

Bitwise	& (and)	(or)	^ (xor)	~ (not)
Assignment	=	&=	=	^=
Equality	==	!=		

```
sc_bit a, b; // declaration  
a = a & b;  
a = a | b;
```

$a \&= b \Rightarrow a = a \& b$

Type `sc_logic`

Example 1

- Four-valued '0' (false), '1' (true), 'X' (unknown) and 'Z' (high impedance)
- Used to model multi-driver buses, X propagation, startup values, and floating buses
- Used on RTL simulation

Bitwise	& (and)	(or)	^ (xor)	~ (not)
Assignment	=	&=	=	^=
Equality	==	!=		

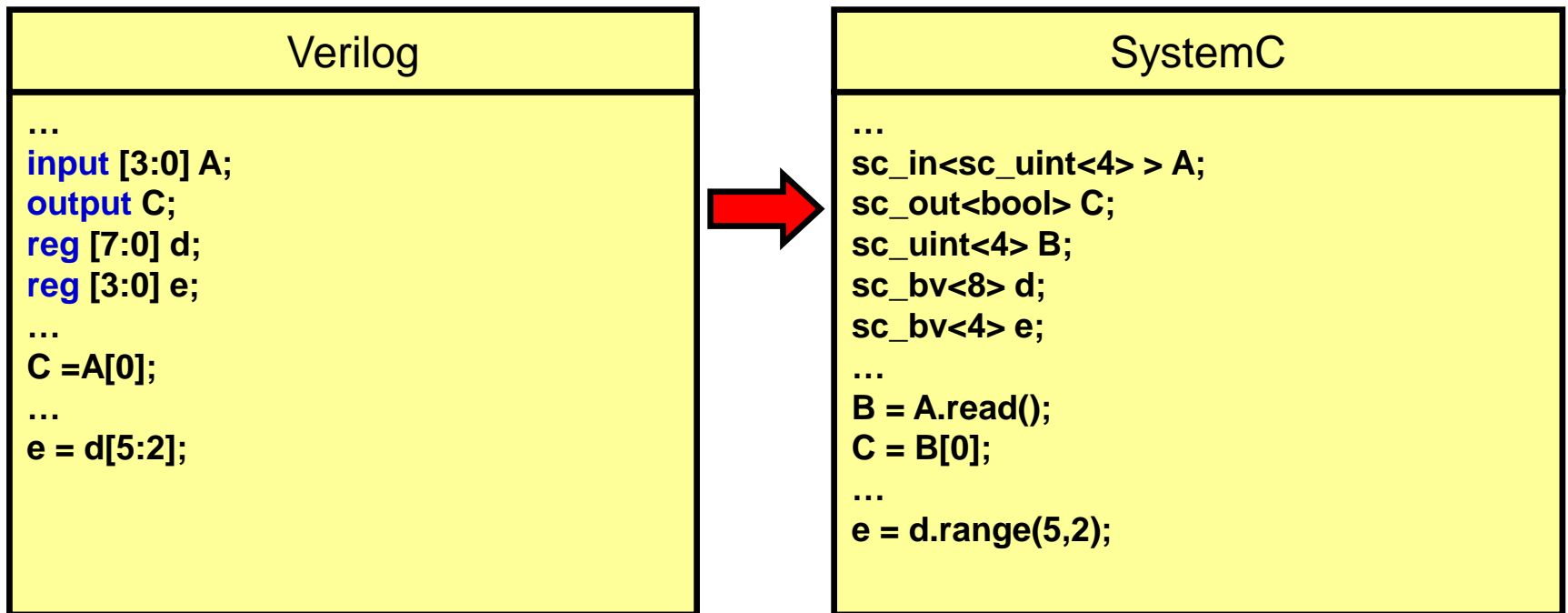
```
sc_logic x; // declaration
x = '1';
x = 'Z';
```

Fixed Precision Integer

- C++ int type: usually 32 bits
- SystemC: 1 to 64 bits
 - ◆ `sc_int<n>`, `sc_uint<n>`

Bitwise	<code>~</code> <code>&</code> <code> </code> <code>^</code> <code>>></code> <code><<</code>
Arithmetic	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>
Assignment	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&=</code> <code> =</code> <code>^=</code>
Equality	<code>==</code> <code>!=</code>
Relational	<code><</code> <code><=</code> <code>></code> <code>>=</code>
Autoincrement	<code>++</code>
Autodecrement	<code>--</code>
Bit select	<code>[x]</code>
Part select	<code>range()</code>
Concatenation	<code>(,)</code>

Bit Select and Part Select



Arbitrary Precision Integer

- `sc_biguint`, `sc_bigint` can larger than 64 bits, limited only by system
- Execute more slowly
- No concatenation

Bitwise	<code>~</code> <code>&</code> <code> </code> <code>^</code> <code>>></code> <code><<</code>
Arithmetic	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>
Assignment	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&=</code> <code> =</code> <code>^=</code>
Equality	<code>==</code> <code>!=</code>
Relational	<code><</code> <code><=</code> <code>></code> <code>>=</code>
Autoincrement	<code>++</code>
Autodecrement	<code>--</code>
Bit select	<code>[x]</code>
Part select	<code>range()</code>

Arbitrary Length Bit Vector

- `sc_bv<n>` is arbitrary length (n) vector of `sc_bit`
- `sc_bv` simulation faster than `sc_lv`

Bitwise	<code>~</code> <code>&</code> <code> </code> <code>^</code> <code>>></code> <code><<</code>
Assignment	<code>=</code> <code>&=</code> <code> =</code> <code>^=</code>
Equality	<code>==</code> <code>!=</code>
Bit select	<code>[x]</code>
Part select	<code>range()</code>
Concatenation	<code>(,)</code>
Reduction	<code>and_reduction()</code> <code>or_reduction()</code> <code>xor_reduction()</code>

```
sc_bv<64> databus;  
sc_logic result;  
result = databus.or_reduction();
```

Arbitrary Length Logic Vector

- `sc_lv<n>` is arbitrary length (n) vector of `sc_logic`
- Help to model tri-state capabilities

Bitwise	& (and)	(or)	^ (xor)	~ (not)
Assignment	=	&=	=	^=
Equality	==	!=		

```
sc_uint<16> uint16;  
sc_int<16> int16;  
sc_lv<16> lv16;  
lv16 = uint16; // convert uint to lv  
int16 = lv16; // convert lv to int  
// assign lv to int to perform arithmetic operation
```

Fixed Point Type

- Use to model the behavior of fixed point hardware
- Use to develop DSP software
- Templated type `sc_fixed` and `sc_ufixed`
 - ◆ Use `static arguments`, known at `compile time`
- Untemplated type `sc_fix` and `sc_ufix`
 - ◆ Arguments can use `variables` to determine

```
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> x;  
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> y;  
sc_fix x(list of options);  
sc_ufix y(list of options);
```

- ◆ `wl` - total word length
- ◆ `iwl` – integer word length
- ◆ `q_mode` – quantization mode
- ◆ `o_mode` – overflow mode
- ◆ `n_bits` – number of saturated bits

Operations of Fixed Point



Bitwise	~ & ^
Arithmetic	+ - * / % >> << ++ --
Assignment	= += -= *= /= %= &= = ^= <<= >>=
Equality	== !=
Relational	< <= > >=

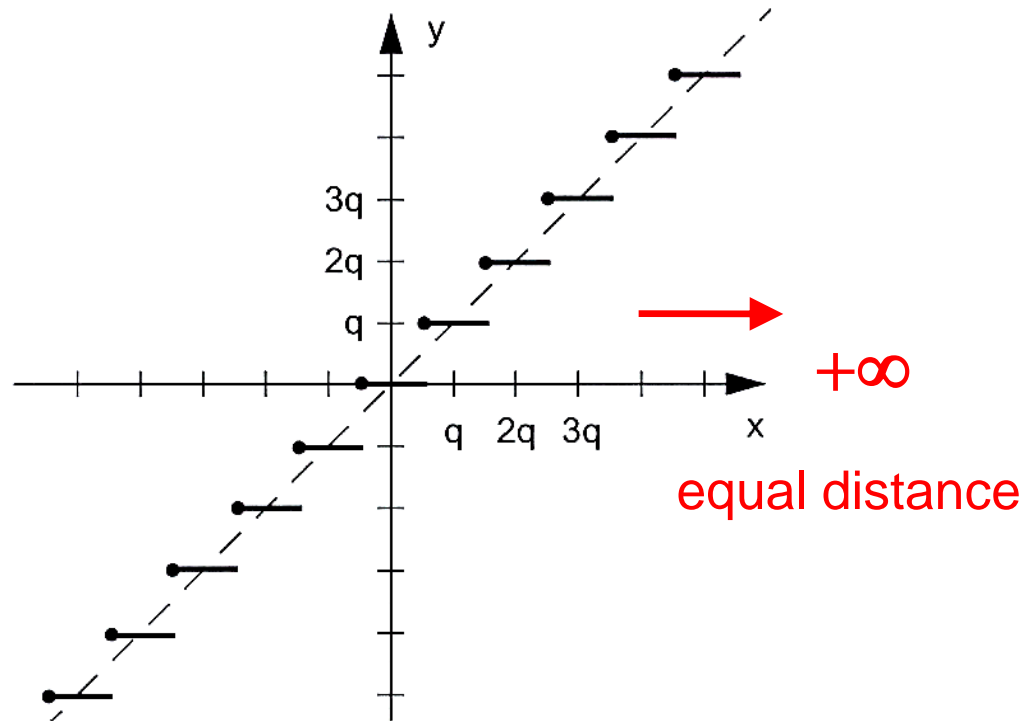
Quantization Modes



Quantization mode	Name
Rounding to plus infinity	SC_RND
Rounding to zero	SC_RND_ZERO
Rounding to minus infinity	SC_RND_MIN_INF
Rounding to infinity	SC_RND_INF
Convergent rounding	SC_RND_CONV
Truncation	SC_TRN
Truncation to zero	SC_TRN_ZERO

SC_RND

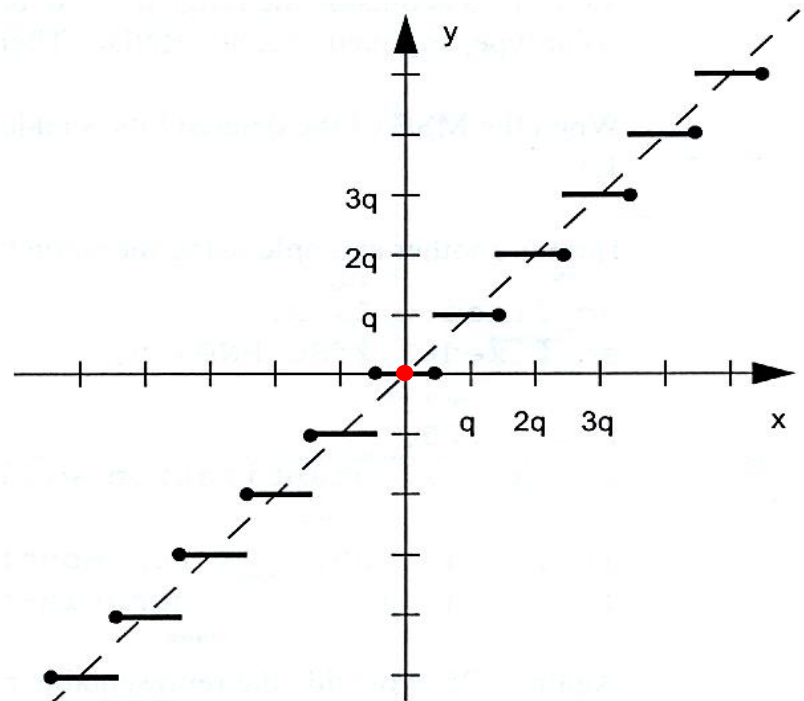
- Round the value to the closest representable number by adding the MSB of the removed bits to the remaining bits
- 01001.010[10100] (x) \rightarrow 01001.011 (y)
- 01.01 (1.25) \rightarrow 01.1 (1.5), 010.1 (2.5) \rightarrow 011 (3)
- 10.11 (-1.25) \rightarrow 11.0 (-1), 101.1 (-2.5) \rightarrow 110 (-2)



[4]

SC_RND_ZERO

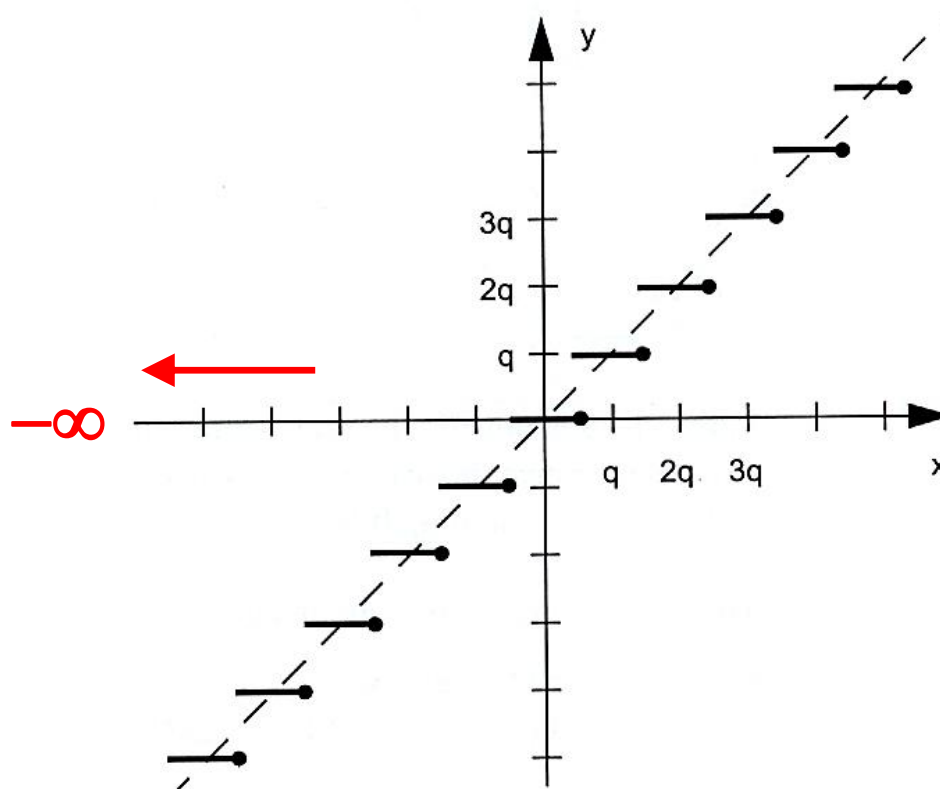
- Perform SC_RND if the two nearest representable numbers are not an **equal distance** apart, otherwise **round to zero** is performed
- Positive numbers: the redundant bits on the LSB side shall be deleted
- Negative numbers: the most significant of the deleted LSBs shall be added to the remaining bits
- 010.1 (2.5) $\rightarrow 010$ (2)
- 101.1 (-2.5) $\rightarrow 110$ (-2)



[4]

SC_RND_MIN_INF

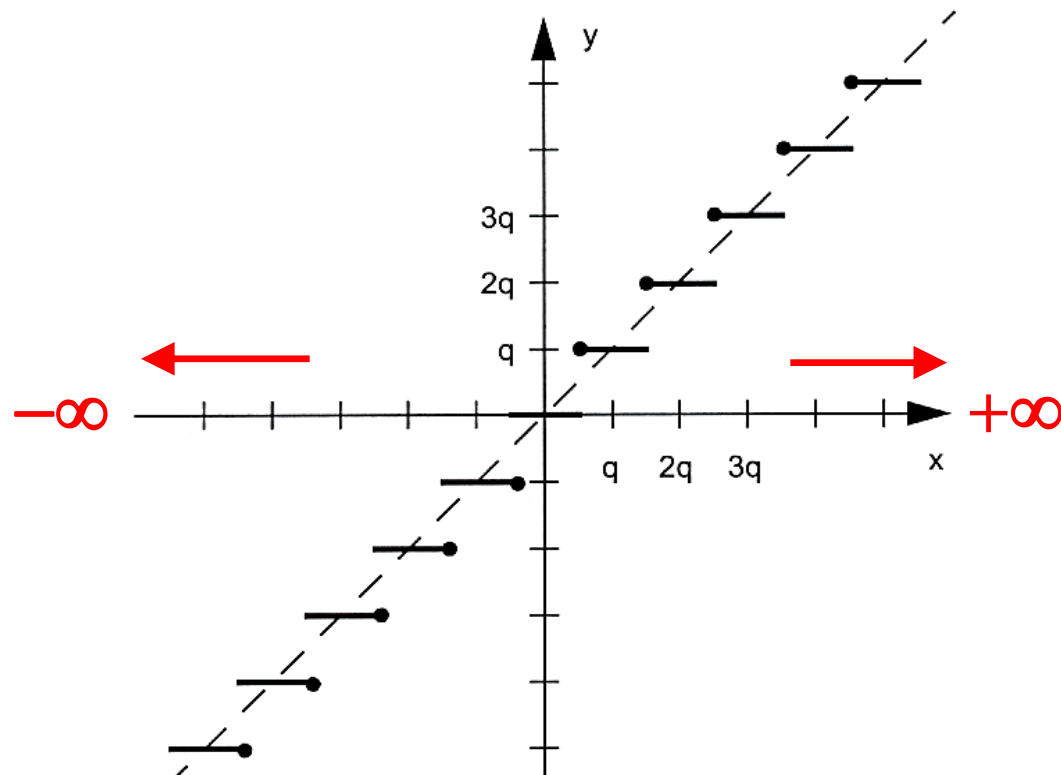
- Performs SC_RND if the two nearest representable numbers are not an equal distance apart, otherwise round to minus infinity is performed



[4]

SC_RND_INF

- Rounding shall be performed if the two nearest representable numbers are at **equal distance**
- Positive numbers: rounding towards **plus infinity**
- Negative numbers: rounding towards **minus infinity**



Overflow Mode



Overflow mode	Name
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical saturation	SC_SAT_SYM
Wrap-around	SC_WRAP
Sign magnitude wrap-around	SC_WRAP_SM

SC_SAT

■ Converts the value to **MAX** for an **overflow** and **MIN** for an **underflow**

■ Ex: `sc_fixed<3,3,SC_TRN,SC_SAT> x;`

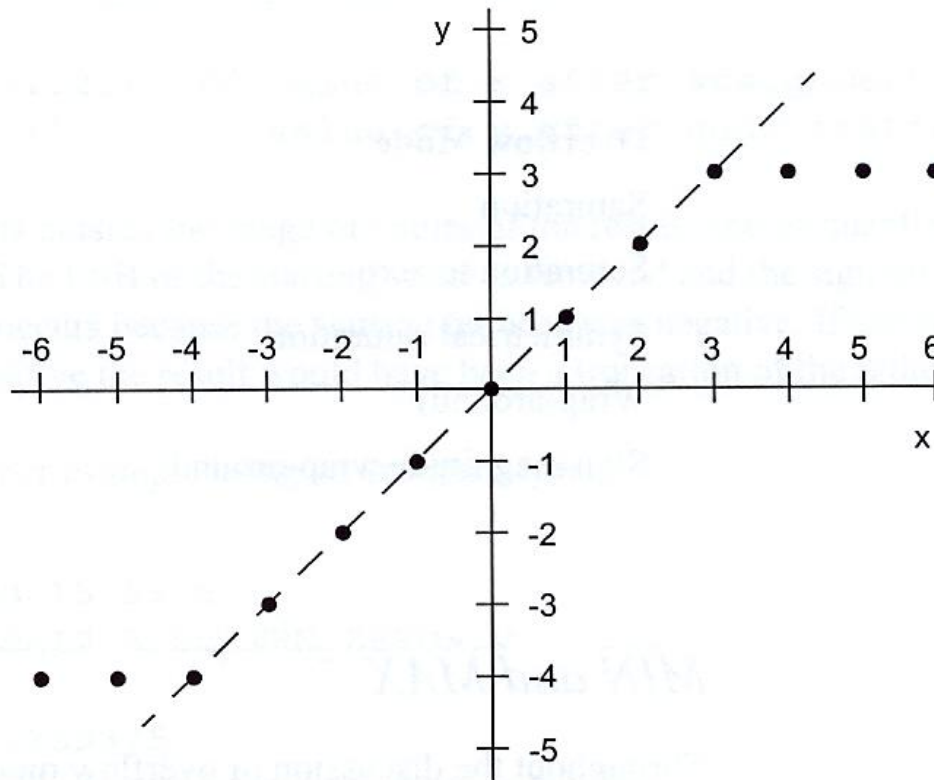
■ 3 bits: MAX (011 = +3), MIN: (100 = -4)

wl - total word length

iwl – interger word length

q_mode – quantization mode

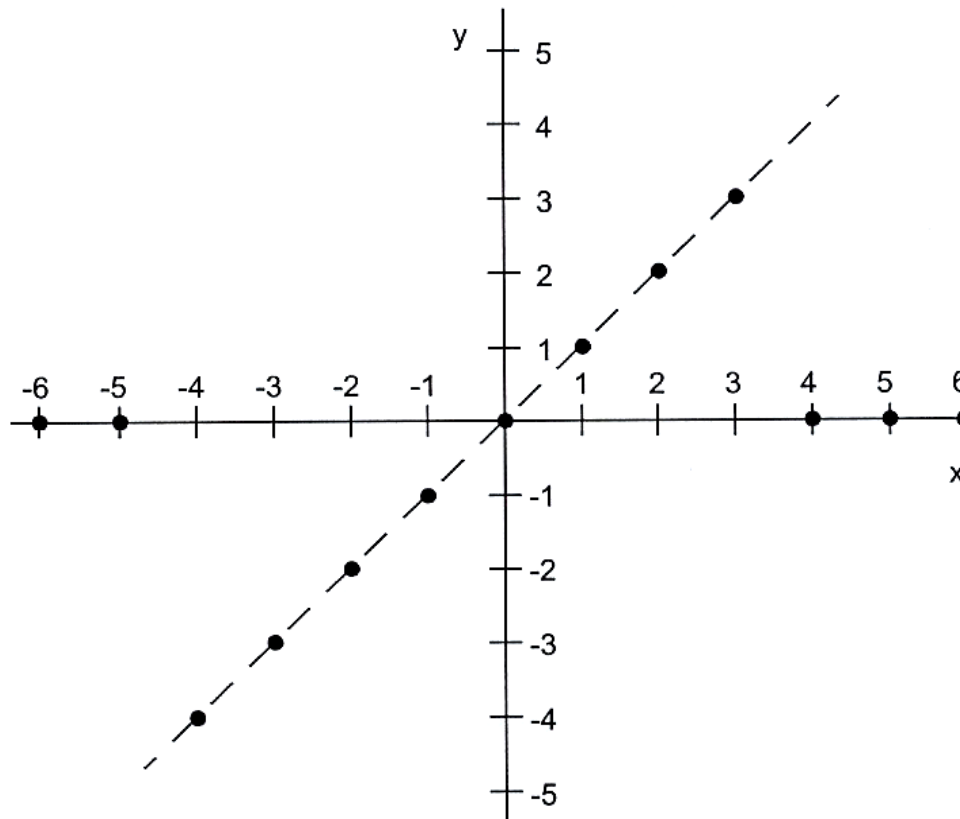
o_mode – overflow mode



SC_SAT_ZERO

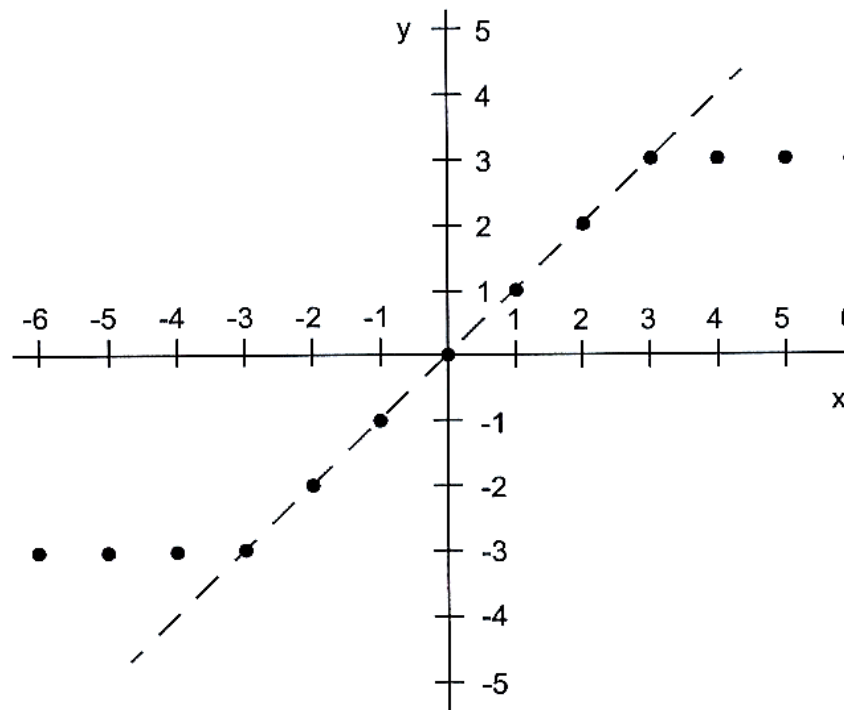


- Sets the result to 0 for any value that is outside the representable range of the fixed point data type



SC_SAT_SYM

- In 2's complement notation there is one more negative representation than positive
- Sometimes it is desirable to have the same positive and negative numbers of representations
- the output is saturated to **MAX** in case of **overflow**, to **-MAX** (signed) or MIN (unsigned) in the case of **negative overflow**



[4]

Time



■ Data type `sc_time`

◆ Time unit:

```
enum sc_time_unit {  
    SC_FS = 0,    // femtosecond  
    SC_PS,        // picosecond  
    SC_NS,        // nanosecond  
    SC_US         // microsecond  
    SC_MS,        // millisecond  
    SC_SEC        // sec };
```

◆ `sc_time` `t(123, SC_NS);` // `t` = 123 nanoseconds

Time Resolution



- The **time resolution** is the **smallest amount of time** that can be represented by all **sc_time** objects in a SystemC simulation
 - ◆ the default value for the time resolution is **1 picosecond**
- A user may set the time resolution to some other value by calling the **sc_set_time_resolution()**
 - ◆ this function, if called, must be called before any **sc_time** objects are constructed
- A user may ascertain the current time resolution by calling the **sc_get_time_resolution()**
- Any time smaller than the time resolution will be rounded off, using **round-to-nearest**

Time Unit



- Time unit is to specify the period of one simulation step
 - ◆ it is larger or equal to time resolution
 - ◆ the default time unit is 1 nanosecond
- Call `sc_set_default_time_unit()` to set default time unit
 - ◆ `sc_set_default_time_unit(10, SC_MS)`
- Time values may sometimes be specified with a numeric value without time unit:
 - ◆ `sc_start(1000);` // run for 1000 time units

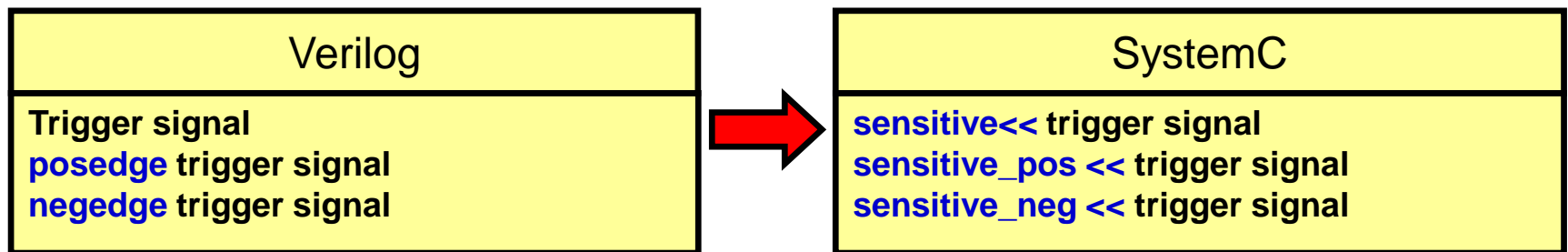
Construct Model



- Module
- Port, interface and channel
- Data type
- **Process**

Processes

- Basic execution unit in SystemC module
- Must be contained in a module
- Can't be hierarchical, can't call other processes
- Can call methods and functions not belong to other processes
- Classify:
 - ◆ SC_METHOD: method
 - ◆ SC_THREAD: thread
 - ◆ SC_CTHREAD: clocked thread
- Have sensitivity list: to be triggered by signal or event



SC_METHOD (1/2)

- Similar to the Verilog `always@` block
- It is called based on the `dynamic` or `static` sensitivity
- A locally declared variable is `not` permanent
- Example:

Static sensitivity:

```
sc_in_clk iclk;  
SC_CTOR(module_name)  
{  
    SC_METHOD(method_name);  
    sensitive<<iclk.neg();  
}
```

Dynamic sensitivity:

```
void method_name(void)  
{  
    //statements  
    next_trigger(time or events)  
}
```


SC_METHOD (2/2)

- Executed when events occur on the sensitivity list
- If execution finish, return control back to simulation kernel
- Recommend **not** to write **infinite loop** within method process
- Can't call **wait()**

```
SC_MODULE ( rcv ){  
    sc_in< frame_type > xin;  
    sc_out < int > id;  
    void extract_id();  
  
    SC_CTOR( rcv ){  
        SC_METHOD(extract_id);  
        //register member function  
        //with simulation kernel  
        sensitive(xin);  
    }  
}
```

rcv.h

```
void rcv::extract_id(){  
    frame_type frame;  
    frame = xin;  
    if(frame == 1){  
        id = frame.ida;  
    }  
    else{  
        id = frame.idb;  
    }  
}
```

rcv.cpp

Example: Hello World

```
#include <systemc.h>
```

```
#include <iostream>
```

```
SC_MODULE(Hello_SystemC)
```

```
{  
  sc_in_clk iclk;
```

Input clock

```
  SC_CTOR(Hello_SystemC)  
  {
```

Constructor

```
    SC_METHOD(method1);
```

```
    sensitive<<iclk.pos();
```

Sensitive to
posedge clock

```
    dont_initialize();
```

```
    SC_METHOD(method2);
```

```
    sensitive<<iclk.neg();
```

```
    dont_initialize();
```

```
  }
```

```
  void method1(void)
```

```
  {
```

```
    std::cout<<sc_time_stamp()
```

get simulation
time

```
        <<" Hello world1!"
```

```
        <<std::endl;
```

```
  }
```

```
void method2(void)
```

```
{
```

```
  std::cout<<sc_time_stamp()
```

```
    <<" Hello world2!"
```

```
    <<std::endl;
```

```
}
```

```
};
```

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

```
{
```

```
  const sc_time t_PERIOD(8,SC_NS);
```

Create clock
signal

```
  sc_clock clk("clk",t_PERIOD);
```

```
  Hello_SystemC iHelloWorld("iHelloWorld");
```

```
  iHelloWorld.iclk(clk);
```

Instantiate
the module

```
  sc_start(20,SC_NS);
```

Port connection

```
  return 0;
```

Start the simulation
with 20ns

```
}
```

The Result



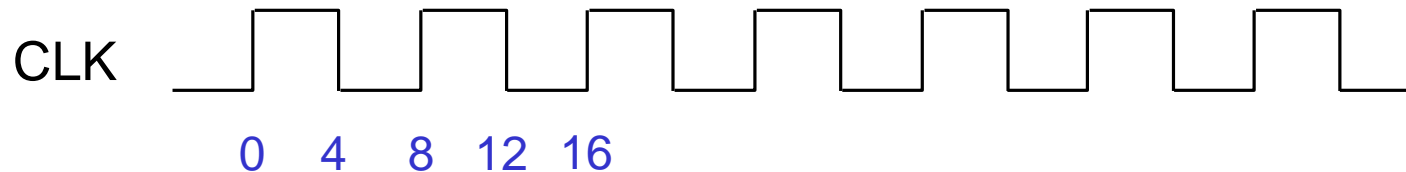
0 ns Hello world1!

4 ns Hello world2!

8 ns Hello world1!

12 ns Hello world2!

16 ns Hello world1!



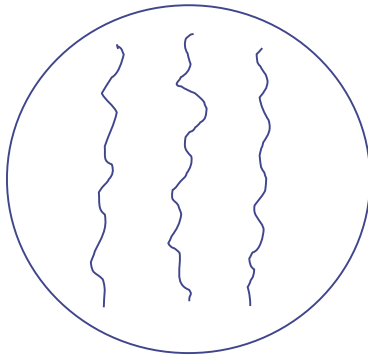
SC_THREAD (1/3)



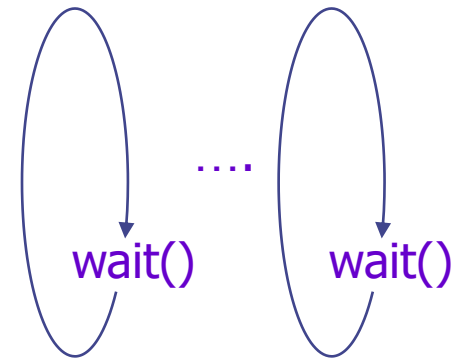
- Once start to execute, it is in complete control of the simulation
- When terminated, it is gone forever
- Typically, it contains a *infinite loop* and at least a *wait*
- Its local variables are alive throughout the simulation
- SC_CTHREAD is a small variation of SC_THREAD
 - ◆ It is triggered at every clock edge

SC_THREAD (2/3)

- The simulation kernel is occupied by the thread in execution and released when a `wait()` is encountered in the thread
- As a whole, the simulation kernel switches in between threads like that in OS



Threads in OS



Threads in SystemC

SC_THREAD (3/3)

- Can be suspended and reactivated by **wait()** call
- Usually contain **infinite loop**, process continue to execute until the **next wait()**

```
SC_MODULE ( rcv ){  
    sc_in< frame_type > xin;  
    sc_out < int > id;  
    void extract_id();  
  
    SC_CTOR( rcv ){  
        SC_THREAD(extract_id);  
        //register member function  
        //with simulation kernel  
        sensitive<<xin;  
    }  
}
```

rcv.h

```
void rcv::extract_id(){  
    frame_type frame;  
    while(true){ //infinite loop  
        frame = xin;  
        if(frame == 1){  
            id = frame.ida;  
        }  
        else{  
            id = frame.idb;  
        }  
        wait();  
        // use wait() call to suspend process  
        // until xin signal change  
    }  
}
```

rcv.cpp

- Only triggered on one of **clock edge** (positive or negative)
- Used to create implicit state machine
 - ◆ States not defined explicitly
 - ◆ States transition by insert **wait()** calls

```
SC_MODULE ( plus ){  
    sc_in_clk clk;  
    sc_in< int > i1;  
    sc_in< int > i2;  
    sc_out < int > o1;  
    void do_plus();  
  
    SC_CTOR( plus ){  
        SC_CTHREAD(do_plus, clk.pos() );  
        // use clk.pos() indicate to trigger  
        // on positive edge  
    }  
}
```

plus.h

```
void plus::do_plus(){  
    int arg1, arg2, sum;  
    while(true){ //infinite loop  
        arg1 = i1.read();  
        arg2 = i2.read();  
        sum = arg1 + arg2;  
        o1.write(sum);  
        wait();  
        // use wait() call to suspend process  
        // until clk.pos signal change  
    }  
}
```

plus.cpp

Sensitivity



- Processes be resumed or activated by sensitivity
 - ◆ Sensitive to **events**
 - ◆ **Ports, signals** have events to trigger implicitly
- Classify:
 - ◆ **Static sensitivity**
 - ▶ Sensitivity **list determined before simulation begin**
 - ▶ RTL and synchronous behavioral only use static sensitivity
 - ◆ **Dynamic sensitivity**
 - ▶ **Temporarily override its static sensitivity list**
 - ▶ Thread process can suspend itself during simulation
 - ▶ Static sensitivity list ignored temporarily

Sensitive Semantic



- Sensitive on edge of Boolean port x
 - ◆ Positive: `sensitive_pos<<x`
 - ◆ Negative: `sensitive_neg<<x`
- Sensitive on event of port x
 - ◆ `sensitive<<x`
- Sensitive on event e
 - ◆ `sensitive<<e`

Dynamic Sensitivity



- Sensitive by calling function as process execution
 - ◆ Used on SC_METHOD
 - ▶ `next_trigger()`
 - ◆ Used on SC_THREAD
 - ▶ `wait()`
 - ◆ Used on SC_CTHREAD
 - ▶ `wait_until()`, `watching()`

Dynamic Sensitivity – next_trigger()

■ Used on SC_METHOD

- ◆ Process invoked only when next_trigger() event occurred

```
SC_MODULE ( rcv ){  
    sc_in< frame_type > xin;  
    sc_out < int > id;  
    void extract_id();  
  
    SC_CTOR( rcv ){  
        SC_METHOD(extract_id);  
        //register member function  
        //with simulation kernel  
        sensitive(xin);  
    }  
}
```

rcv.h

```
void rcv::extract_id(){  
    frame_type frame;  
    sc_event E;  
  
    frame = xin;  
    if(frame == 1){  
        id = frame.ida;  
    }else{  
        id = frame.idb;  
    }  
    next_trigger(200, SC_NS, E);  
    // during 200ns, process can be invoked  
    // by E event; if timeout, invoked by xin  
}
```

rcv.cpp

Dynamic Sensitivity – wait()

■ Used on SC_THREAD

- ◆ Thread suspend when call wait(E), and resumed when E occurred

```
SC_MODULE ( rcv ){  
    sc_in< frame_type > xin;  
    sc_out < int > id;  
    void extract_id();  
  
    SC_CTOR( rcv ){  
        SC_THREAD(extract_id);  
        //register member function  
        //with simulation kernel  
        sensitive<<xin;  
    }  
}
```

rcv.h

```
void rcv::extract_id(){  
    frame_type frame;  
    while(true){ //infinite loop  
        frame = xin;  
        if(frame == 1){  
            id = frame.ida;  
        }else{  
            id = frame.idb;  
        }  
        wait(200, SC_NS, E);  
        // during 200ns, thread suspend on wait()  
        // call; thread resumed when timeout or  
        // E occurred  
    }  
}
```

rcv.cpp

Dynamic Sensitivity – wait_until()

■ Used on SC_CTHREAD

- ◆ Halt execution until event occurred

```
SC_MODULE ( plus ){  
    sc_in_clk clk;  
    sc_in< int > i1;  
    sc_in< int > i2;  
    sc_in< bool > sensor;  
    sc_out < int > o1;  
    void do_plus();  
  
    SC_CTOR( plus ){  
        SC_CTHREAD(do_plus, clk.pos() );  
        // use clk.pos() indicate to trigger  
        // on positive edge  
    }  
}
```

plus.h

```
void plus::do_plus(){  
    int arg1, arg2, sum;  
    while(true){ //infinite loop  
        arg1 = i1.read();  
        arg2 = i2.read();  
        sum = arg1 + arg2;  
        o1.write(sum);  
        wait_until(sensor.delayed() == true);  
        // halt execution until new value of  
        // sensor is true  
    }  
}
```

plus.cpp

Dynamic Sensitivity – watching()

■ Used on SC_CTHREAD

- ◆ Monitor the specified condition
- ◆ Halt current execution, and restart the execution from the first line of process

```
SC_MODULE ( plus ){  
    sc_in_clk clk;  
    sc_in< int > i1;  
    sc_in< int > i2;  
    sc_in< bool > reset;  
    sc_out< int > o1;  
    void do_plus();  
  
    SC_CTOR( plus ){  
        SC_CTHREAD(do_plus, clk.pos() );  
        watching(reset.delayed() == true);  
        // if reset change to true, scheduler  
        // halt execution, and start from the  
        // first line of process  
    }  
}
```

plus.h

```
void plus::do_plus(){  
    int arg1, arg2, sum;  
    if(reset == true){  
        o1.write(0);  
    }  
    while(true){ //infinite loop  
        arg1 = i1.read();  
        arg2 = i2.read();  
        sum = arg1 + arg2;  
        o1.write(sum);  
        wait();  
    }  
}
```

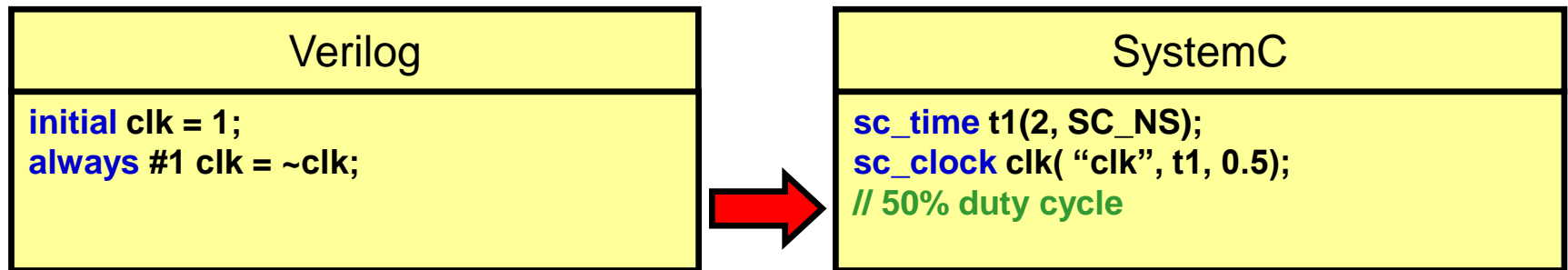
plus.cpp

Comparison on Processes

	SC_METHOD	SC_THREAD	SC_CTHREAD
Execution	When trigger	Always execute	Always execute
Suspend & resume	No	Yes	Yes
Static sensitivity	By sensitive list	By sensitive list	By signal edge
Dynamic sensitivity	next_trigger()	wait()	wait_until(), watching()
Applied model	RTL, synchronize	Behavioral	Clocked behavior

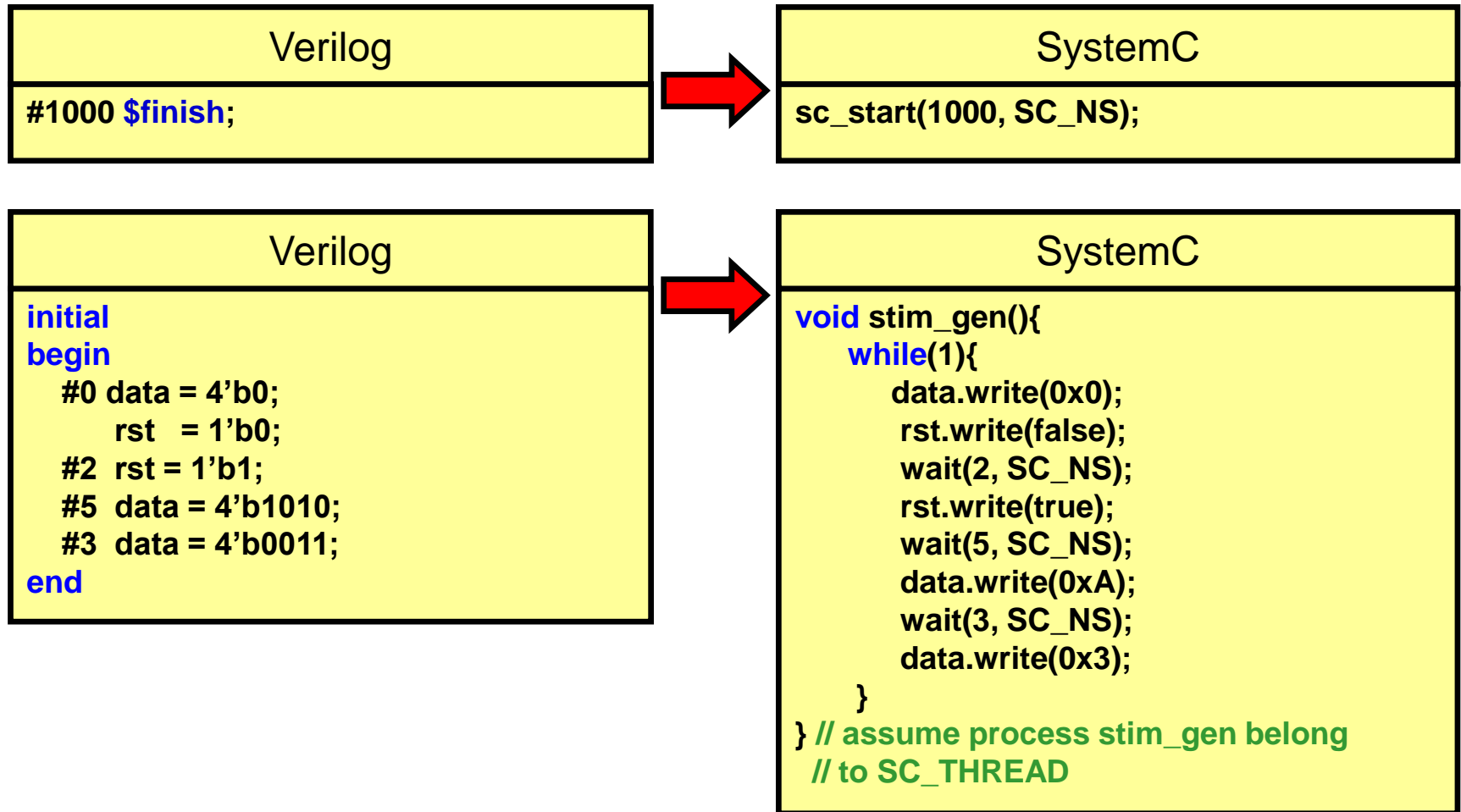
Clock Object

- Generate timing signals to synchronize events
- Typically created at top level in the testbench



```
int sc_main(int argc, char* argv){  
    sc_signal <int> val;  
    sc_signal <sc_logic> reset  
    sc_signal <int> result;  
    sc_time t1(20, SC_NS);  
    sc_clock clk1 ("clk1", t1, 0.5, 0, true);  
    // create a clock clk1 with period 20ns, duty cycle 50%, first edge at time 0  
    // and first value is true  
    filter f1("filter");  
    f1.clk(clk1.signal());  
    f1.val(val);  
    f1.reset(reset);  
    f1.result(result);  
}
```


Simulation Control

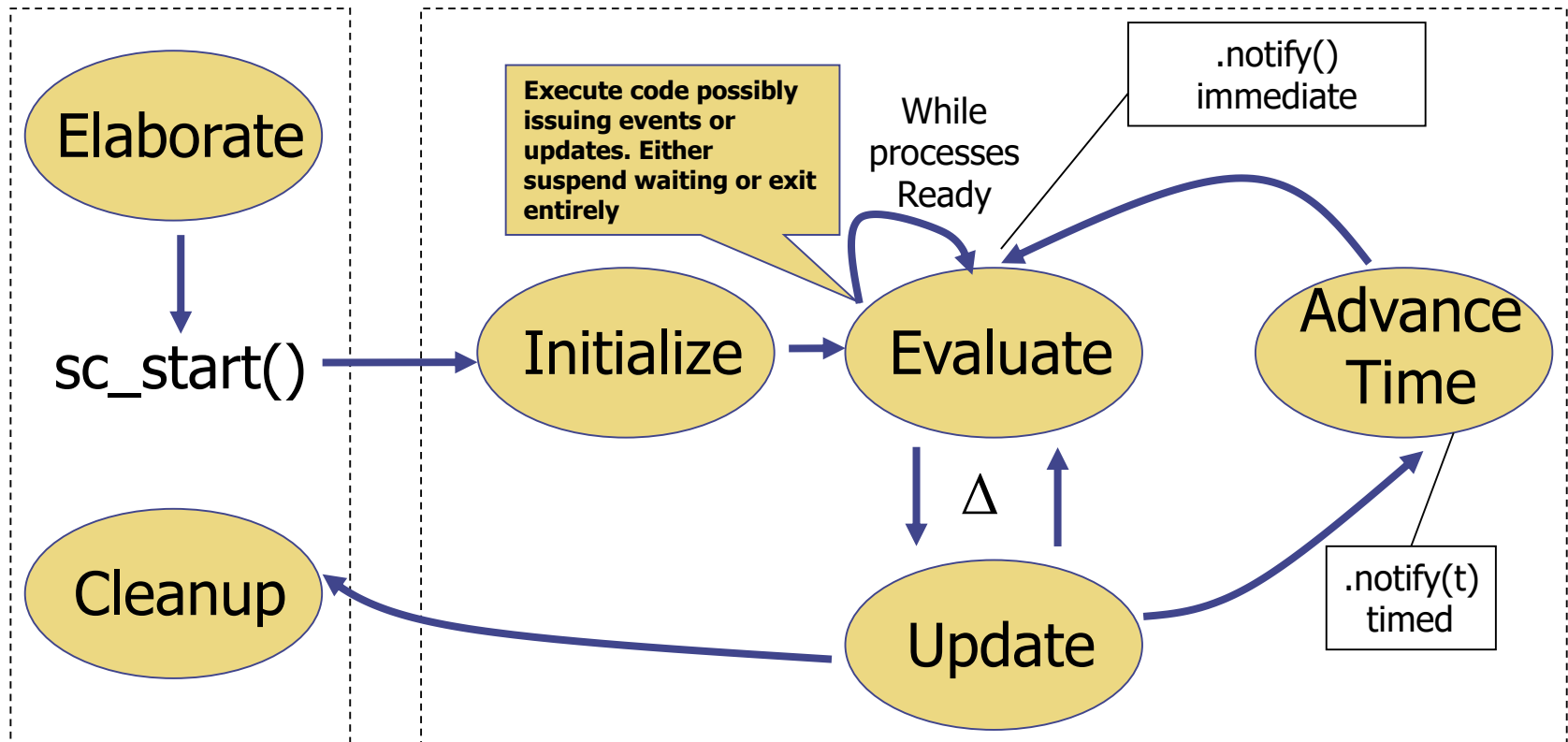


SystemC Simulation Kernel



sc_main()

SystemC Simulation Kernel



[1]

Simulation Semantics



- The **scheduler** controls the timing and order of process execution, handles event notifications and manages updates to channels
- The scheduler supports the notion of **delta-cycle**, which consists of an evaluate phase and update phase
- Processes are non-preemptive, meaning for a **thread** process, codes between two wait() will **execute without any other process interruption** and a **method** process completes its execution **without interrupted** by another process
- **Elaboration**: The execution of the sc_main() from the start to the first invocation of sc_start()

Initialization Phase



- The first step in the simulator scheduler
 - ◆ Each `method process` is executed once during initialization and each `thread process` is executed until a wait statement is encountered
- To **turn off initialization** for a particular process, call **`dont_initialize()`** after the `SC_METHOD` or `SC_THREAD` process declaration inside a module constructor (**P. 66**)
- The order of processes' execution is unspecified, however the execution order between processes is determined
 - ◆ This only means every two simulation runs to the same code always have the same execution ordering to yield identical results.

Evaluate Phase



- (1) From the set of processes that are ready to run, **select a process and resume its execution**
 - ◆ The order in which processes are selected for execution from the set of processes that are ready to run is unspecified.
- (2) The execution of a process may include calls to the **request_update() function** which schedules pending calls to update() function in the **update phase**
 - ◆ The request_update() function may only be called inside member functions of a primitive channel
- (3) **Repeat evaluation** for any other processes that are **ready** to run - **synchronization**

Update Phase



- (1) Execute any pending calls to update() from calls to the request_update() function executed in the evaluate phase
- (2) If there are pending delta-delay notifications, **determine which processes are ready to run** and go to evaluation phase
- (3) If there are **no more timed event notifications**, the simulation is **finished**
- (4) Else, **advance the current simulation time** to the time of the earliest (next) pending timed event notification
- (5) Determine which processes become ready to run due to the events that have pending notifications at the current time, go to evaluation phase

Debugging Issues



- Basic debugging tools: printf(), breakpoint, variable watching, and waveform tracing
- Don't know how to debug a thread? -> use conditional breakpoint
- Unable to get the value of a SystemC data type? -> use conversion function such as to_int()
- Unable to get the value of a port? -> use my_port.read()
- UNKNOWN EXCEPTION OCCURED? -> basic C/C++ run time problems such as:
 - ◆ Array index out of bound
 - ◆ Freeing an empty pointer

Design Granularity



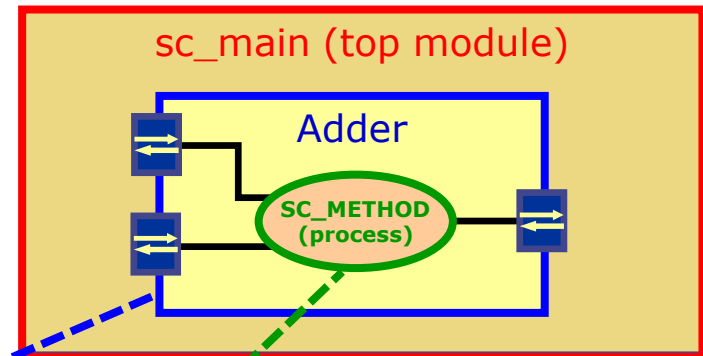
- Use as many C++ primitive data types as possible
- Unless necessary, do **not** model too much detail of your design
 - ◆ This will accelerate both design and simulation
 - ◆ Ex: use **events** rather than **clock triggering**

Examples of SystemC

Summary of System Design with SystemC

■ Pseudo code of templates for

- ◆ `sc_main.cpp`
- ◆ `Module.h`
- ◆ `Module.cpp`
 - ▶ `SC_CTOR`
 - ▶ `SC_HAS_PROCESS`



```
#include <systemc.h>

SC_MODULE(Adder)
{
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> c;

    void compute();

    SC_CTOR(Adder)
    {
        SC_METHOD(compute);
        sensitive << a << b;
    }
};
```

Adder.h

```
#include "Adder.h"

void Adder::compute()
{
    c = a + b;
}
```

Adder.cpp

```
#include <systemc.h>
#include "Adder.h"

int sc_main(int argc, char* argv[])
{
    sc_signal<int> sig_a, sig_b, sig_c;
    Adder my_adder("my_adder");
    my_adder(sig_a, sig_b, sig_c);

    sc_start(1000, SC_SEC);
    return 0;
}
```

main.cpp



```
#include <systemc.h>
#include "module_name.h"

int sc_main(){
    Port/Signal declarations
    module declarations
    modules linked

    sc_start(); /sc_start(time);
    return 0;
}
```

Module.h

SC_CTOR

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
SC_MODULE(NAME) {
    Port declarations
    Channel/submodule instances
    SC_CTOR(NAME)
    : Initializations
    {
        Connectivity
        Process registrations
    }
    Process declarations
    Helper declarations
};
#endif
```

SC_HAS_PROCESS

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
SC_MODULE(NAME) {
    Port declarations
    Channel/submodule instances
    // Constructor declaration:
    NAME();
    Process declarations
    Helper declarations
};
#endif
```



SC_CTOR

```
#include <systemc.h>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

SC_HAS_PROCESS

```
#include <systemc.h>
#include "NAME.h"
NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
    Channel allocations
    Submodule allocations
    Connectivity
    SC_HAS_PROCESS(NAME);
    Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

Example 1: Full-adder

Example 1

```
#include "systemc.h"

SC_MODULE(Adder){
    sc_out<sc_logic> sum, Cout;
    sc_in<sc_logic> A, B, Cin;

    void add(){
        sc_logic tempC, tempD, tempE;
        tempC = A.read() & B.read();
        tempD = A.read() ^ B.read();
        tempE = Cin.read() & tempD;
        sum.write(tempD ^ Cin.read());
        Cout.write(tempC | tempE);
    }

    SC_CTOR(Adder){
        SC_METHOD(add);
        sensitive << A << B << Cin;
    }
};
```

Example 2: Counter (1/2)

Example 2

“counter.h”

```
#include <systemc.h>
SC_MODULE(counter)
{
    // port- declarstions;
    sc_in_clk    clk;
    sc_out<int>  val;

    // process declarations;
    void process_func();

    // module constructor;
    SC_CTOR(counter) {
        SC_CTHREAD(process_func, clk.pos());
    }
};
```

“counter.cpp”

```
#include “counter.h”
counter::process_func()
{
    val = 0;
    while(1)
    {
        wait();
        val = val+1;
    }
}
```

Example 2: Counter (2/2)

“main.cpp”

```
#include "counter.h"
int sc_main(int argc, char* argv[])
{
    // signal declaration
    sc_clock      clk("clk", 1, SC_NS, 0.5);
    sc_signal<int> val;

    // module declaration
    counter      counter0("counter0");

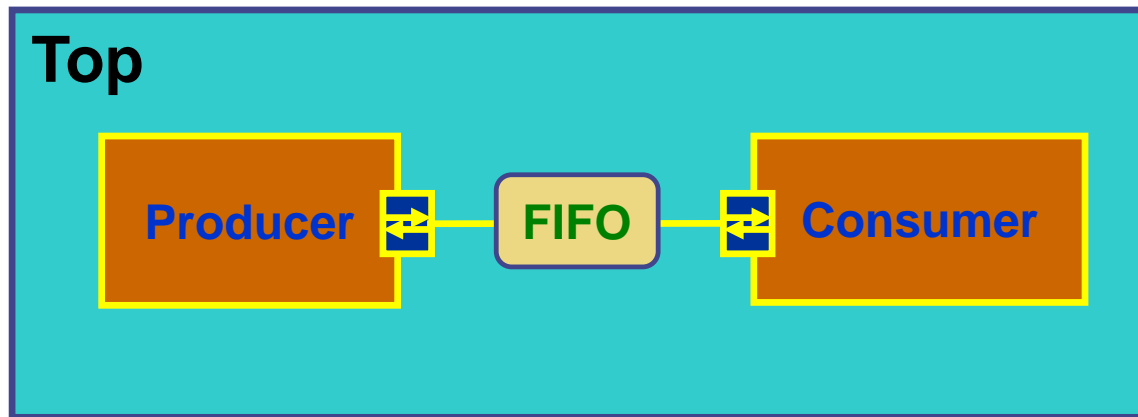
    // signal connection
    counter0.clk(clk);
    counter0.val(val);

    // run simulation
    sc_start(100, SC_NS);

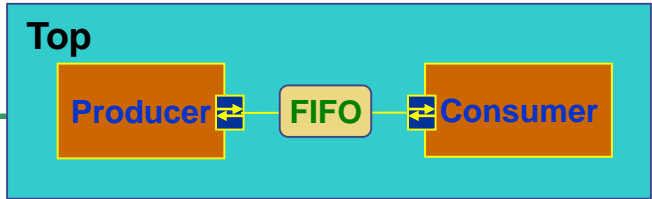
    return 0;
}
```


Example 3: Producer and Consumer (1/3)

Example 3



Producer and Consumer (2/3)



```

#include <systemc.h>
#include <iostream.h>
#include <stdlib.h>
SC_MODULE(consumer)
{
    sc_in_clk iclk;
    sc_fifo_in<int> data_in;
    int data_value;
    SC_CTOR(consumer)
    {
        SC_CTHREAD(x_consumer, iclk.pos());
        data_value=0;
    }
    void x_consumer()
    {
        for(;;){
            wait(3);
            data_value=data_in.read();
            cout<<"At time"<<sc_time_stamp()<<
            "consume data"<<data_value<<endl;
        }
    }
};

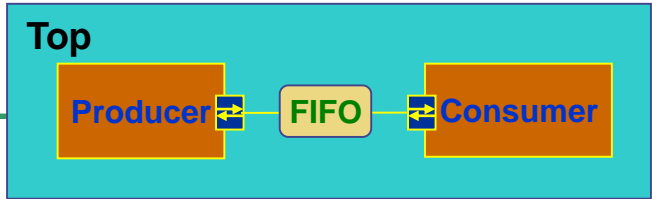
```

```

SC_MODULE(producer)
{
    sc_in_clk iclk;
    sc_fifo_out<int> data_out;
    int data_value;
    SC_CTOR(producer)
    {
        SC_CTHREAD(x_producer, iclk.pos());
        data_value=0;
    }
    void x_producer()
    {
        for(;;){
            wait(2);
            data_value=rand()%100;
            data_out.write(data_value);
            cout<< "At time"<<sc_time_stamp()<<
            "produces data"<<data_value<<endl;
        }
    }
};

```

Producer and Consumer (3/3)



```
int sc_main(int argc, char *argv[])
{
    const sc_time t_PERIOD(10,SC_NS);
    sc_clock clk("clk",t_PERIOD);
    sc_fifo<int> x_fifo;

    producer x_producer("x_producer");
    consumer x_consumer("x_consumer");

    x_producer.iclk(clk);
    x_consumer.iclk(clk);
    x_producer.data_out(x_fifo);
    x_consumer.data_in(x_fifo);
    sc_start(200,SC_NS);
    return 0;
}
```

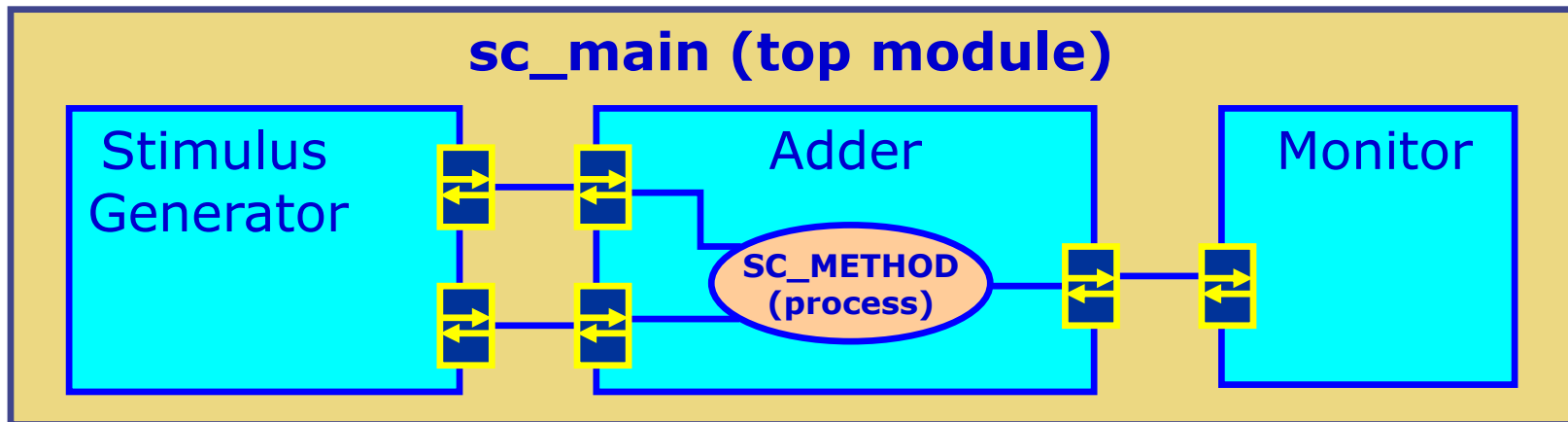
Results

At time 20 ns produces data 38
At time 30 ns consumes data 38
At time 40 ns produces data 58
At time 60 ns produces data 13
At time 60 ns consumes data 58
At time 80 ns produces data 15
At time 90 ns consumes data 13
At time 100 ns produces data 51

Example 4: Test in SystemC (1/6)

Example 4

- Stimulus Generator
- Monitor
- Adder



Example 4: Test in SystemC (2/6)

■ Stimulus Generator

- ◆ Use thread to generate stimulus
- ◆ Binding port and `sc_fifo_out_if<T>`, to store stimulus in order

```
#include <systemc.h>

SC_MODULE(stingen)
{
    //sc_out <int> in1, in2;
    sc_port<sc_fifo_out_if<int> > in1, in2;
    int seed;
    void stim_proc();
    SC_CTOR(stingen)
    {
        seed=12;
        SC_THREAD(stim_proc);
    }
};
```

stingen.h

```
#include "stingen.h"

void stingen::stim_proc()
{
    int temp;
    for(int i=0; i <=20; i++)
    {
        temp = seed+1;
        in1 ->write(temp);
        in2 ->write(temp+5);
        seed = (seed+19)%123;
    }
    sc_stop();
}
```

stingen.cpp

seed	12	31	50	69
in1	13	32	51	70
in2	18	37	56	75

Example 4: Test in SystemC (3/6)

■ Monitor

- ◆ Use thread to monitor output from Design Under Test (DUT)
- ◆ Binding port and `sc_fifo_in_if<T>`, to store output in order
- ◆ Use port `re->read()` to pop data from FIFO buffer

```
#include <systemc.h>

SC_MODULE(monitor)
{
    //sc_in<int> re;
    sc_port<sc_fifo_in_if<int> > re;
    void monitor_proc();
    SC_CTOR(monitor)
    {
        SC_THREAD(monitor_proc);
    }
};
```

monitor.h

```
#include "monitor.h"

void monitor::monitor_proc()
{
    while(true)
    {
        cout <<"The reslut of the computation is = "<<re->read()<<endl;
    }
}
```

monitor.cpp

Example 4: Test in SystemC (4/6)

Adder

- ◆ Both stimulus generator and monitor use FIFO interface
- ◆ Adder modify the in/out port interface to connect
- ◆ Use channel's method `write()` and `read()` to access data

```
#include <systemc.h>

SC_MODULE(Adder)
{
    //sc_in<int>    a;
    //sc_in<int>    b;
    //sc_out<int>   c;
    sc_port<sc_fifo_in_if<int> > a;
    sc_port<sc_fifo_in_if<int> > b;
    sc_port<sc_fifo_out_if<int> > c;

    void compute();

    SC_CTOR(Adder)
    {
        SC_THREAD(compute);
    }
};
```

Adder.h

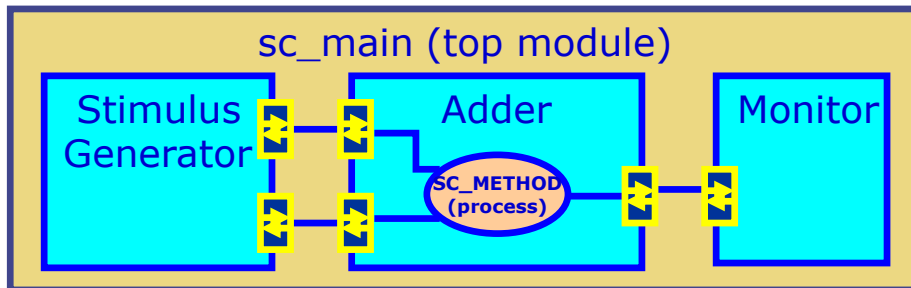
```
#include "Adder.h"

void Adder::compute()
{
    while(true)
    {
        c -> write( a->read() + b->read());
        c -> write( a->read() + b->read()+2);
    }
}
```

Adder.cpp

Example 4: Test in SystemC (5/6)

- `sc_main()`
 - ◆ Program entrance
- `sc_start()`
 - ◆ Simulation begin
 - ◆ `sc_start(-1)` simulate infinitely
 - ◆ `sc_start(100, SC_NS)` simulate for 100ns



```
#include <systemc.h>
#include "Adder.h"
#include "stimgen.h"
#include "monitor.h"

int sc_main(int argc, char* argv[])
{
    //sc_signal<int> sig_a, sig_b, sig_c;
    sc_fifo<int> sig_a(10), sig_b(10), sig_c(10);

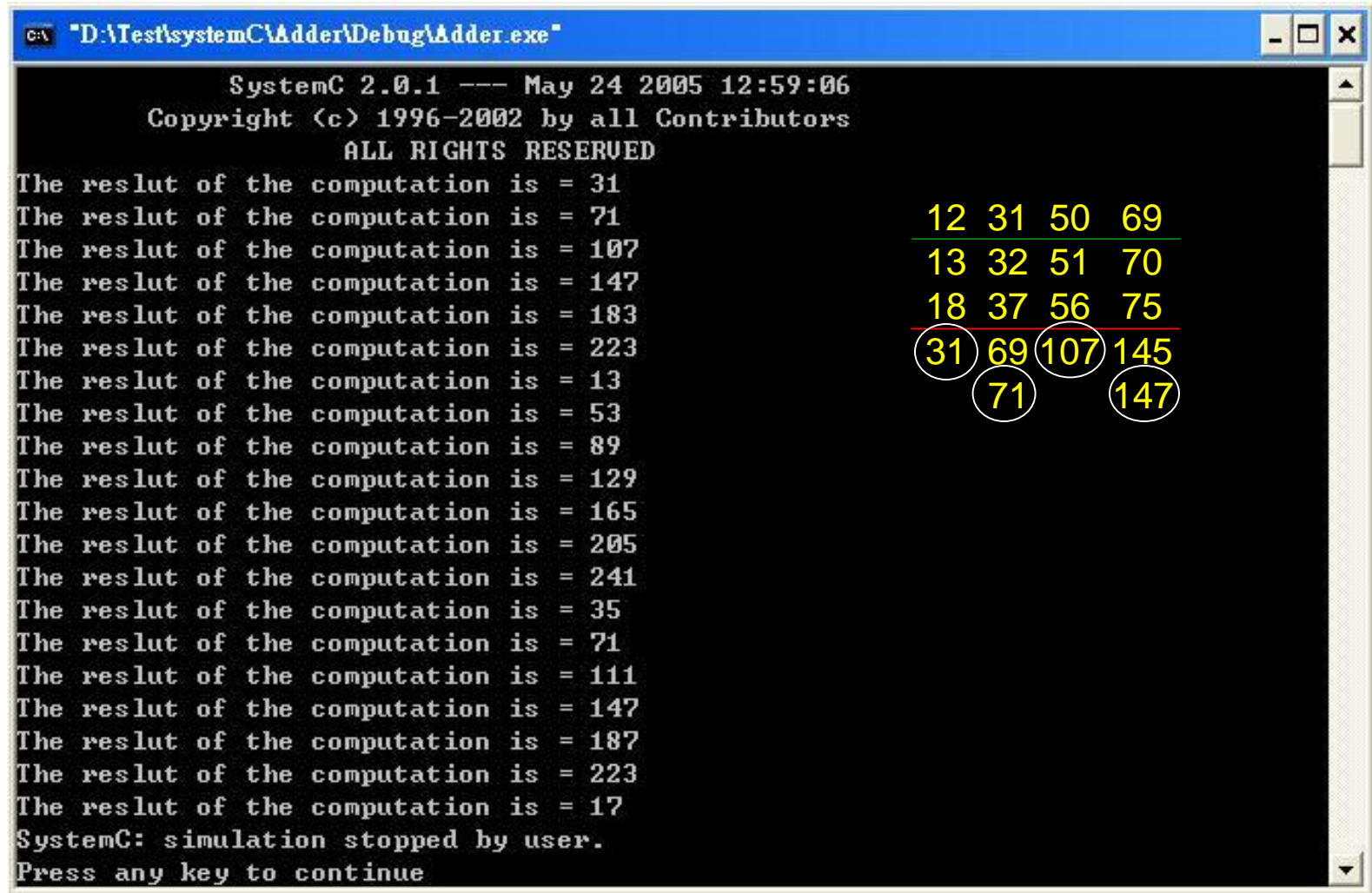
    Adder my_adder("my_adder");
    stimgen stim("stim");

    stim(sig_a, sig_b);
    my_adder(sig_a, sig_b, sig_c);

    monitor mon("mon");
    mon.re(sig_c);

    sc_start();
    return 0;
}
```


Example 4: Test in SystemC (6/6)



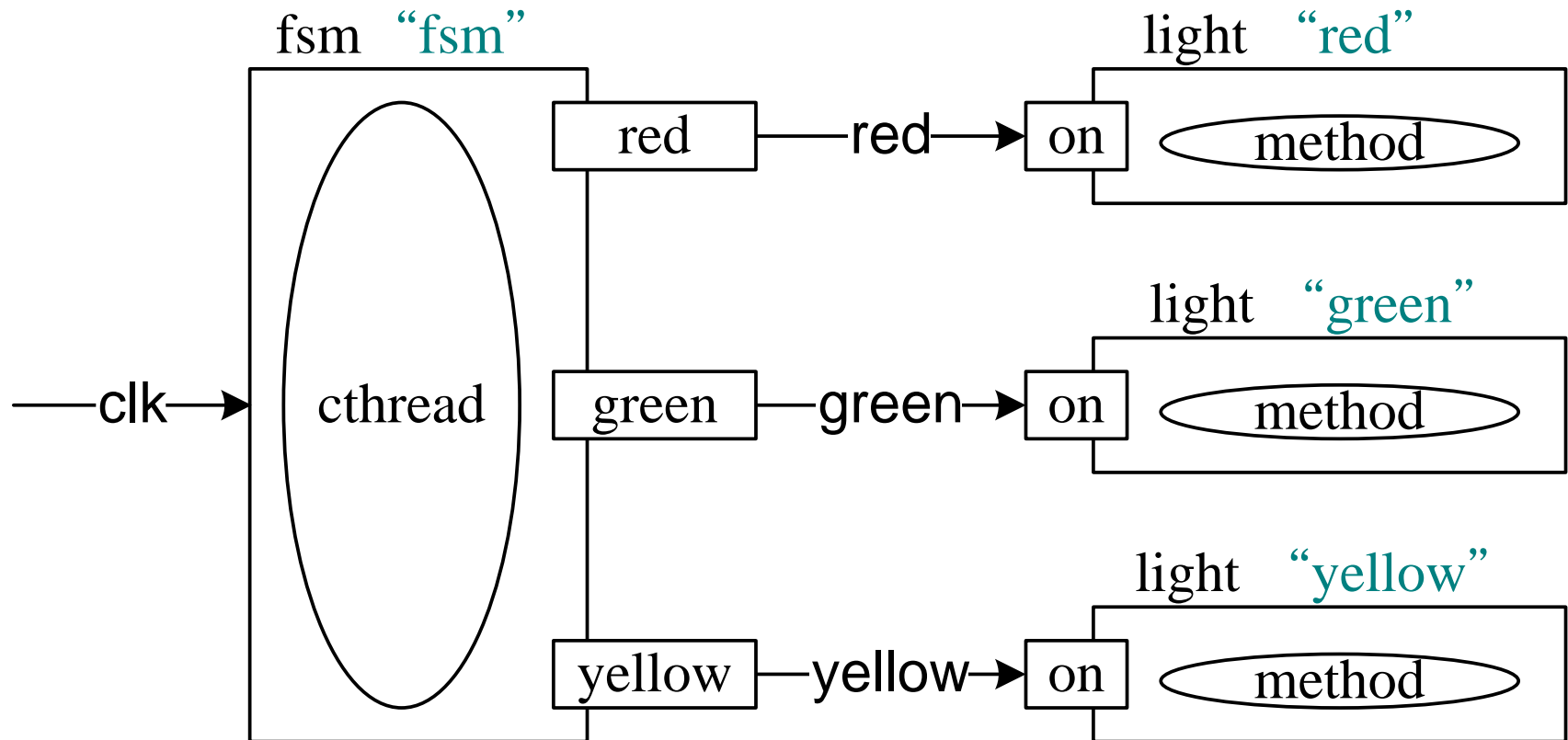
```
C:\ "D:\Test\systemC\Adder\Debug\Adder.exe"

SystemC 2.0.1 --- May 24 2005 12:59:06
Copyright (c) 1996-2002 by all Contributors
ALL RIGHTS RESERVED

The result of the computation is = 31
The result of the computation is = 71
The result of the computation is = 107
The result of the computation is = 147
The result of the computation is = 183
The result of the computation is = 223
The result of the computation is = 13
The result of the computation is = 53
The result of the computation is = 89
The result of the computation is = 129
The result of the computation is = 165
The result of the computation is = 205
The result of the computation is = 241
The result of the computation is = 35
The result of the computation is = 71
The result of the computation is = 111
The result of the computation is = 147
The result of the computation is = 187
The result of the computation is = 223
The result of the computation is = 17
SystemC: simulation stopped by user.
Press any key to continue
```

12	31	50	69
13	32	51	70
18	37	56	75
31	69	107	145
71			147

Example 5: Traffic Light (1/4)



Example 5: Traffic Light (2/4)



■ Finite State Mache

```
void fsm::cthread_func()
{
    while(1) {
        // red
        red=true, green=false, yellow=false;
        wait(10);
        // green
        red=false, green=true, yellow=false;
        wait(10);
        // yellow
        red=false, green=false, yellow=true;
        wait(2);
    }
}
```

Example 5: Traffic Light (3/4)

“light.h”

```
#include <systemc.h>
SC_MODULE(light)
{
    // port declarations
    sc_in<bool>    on;

    // process declarations
    void method_func();

    // constructor
    SC_CTOR(light) {
        SC_METHOD(method_func);
        sensitive << on;
    }
};
```

“light.cpp”

```
#include “light.h”
void light::method_func()
{
    if(on) {
        printf(“%6lld  ps: %s\n”, \
               sc_time_stamp().value(), \
               name())
    }
}
```

Example 5: Traffic Light (4/4)

0 ps: red
10000 ps: green
20000 ps: yellow
22000 ps: red
32000 ps: green
42000 ps: yellow
44000 ps: red
54000 ps: green
64000 ps: yellow
66000 ps: red
76000 ps: green
86000 ps: yellow
88000 ps: red

Example 6: Very Simple Bus (1/2)

- A very simple example is provided to model the bus **burst read** and **burst write** transactions
- This very simple bus model doesn't consider the other important behaviors of the real bus architecture such as bus arbitration, interrupted burst transactions, and memory wait states etc.
- To make the bus simple, memory is modeled as a **memory array** within the bus that the bus can access it directly rather than a memory module external to the bus

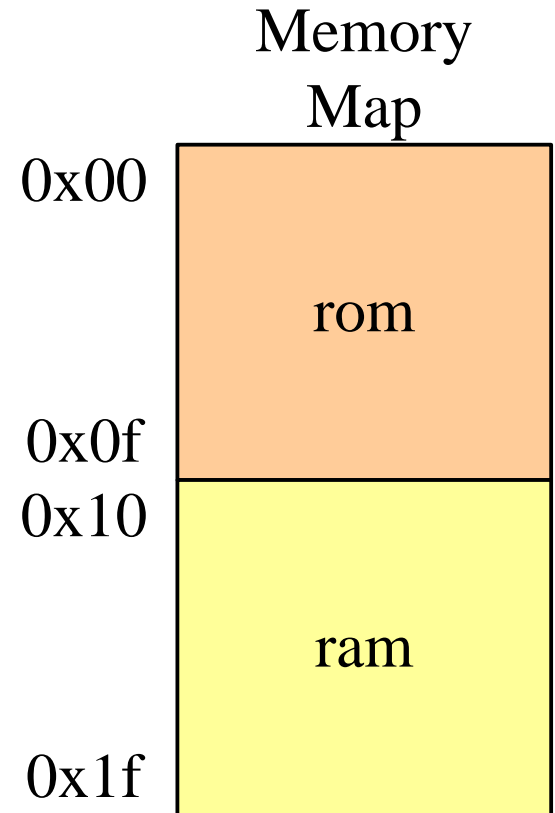
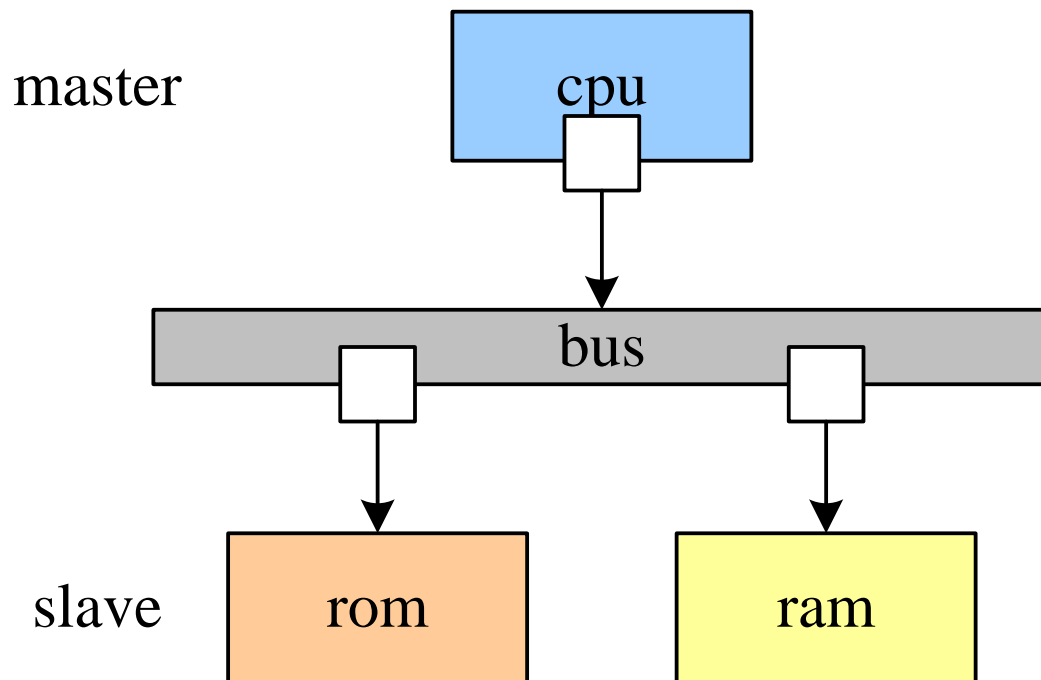
Example 6: Very Simple Bus (2/2)

```
class very_simple_bus_if : virtual public
sc_interface
{
public:
    virtual void burst_write (char *data,
                             unsigned adder, unsigned length)=0;
    virtual void burst_read (char *data,
                             unsigned adder, unsigned length)=0;
};
```

```
class very_simple_bus:
public very_simple_bus_if, public sc_channel
{
public:
    very_simple_bus ( sc_name nm,
                     unsigned mem_size,
                     sc_time cycle_time )
    : sc_channel(nm), _cycle_time(cycle_time)
    {
        //we model bus memory access using an
        //embedded memory array
        _mem = new char [mun_size];
        //set initail value of memory to zero
        memset(_mem, 0, mem_size_);
    }
    ~very_simple_bus() { delet [] _mem;}
```

```
virtual void burst_read (char *data, unsigned adder, unsigned length)
{
    //model bus contention using mutex, but no arbitration rules
    _bus_mux.lock();
    // block the caller for length of burst transaction
    wait (length * _cycle_time);
    // copy the data form memory of burst transaction
    memcpy(data, _mem +addr, length);
    // unlock the mutex to allow others access to the bus
    _bus_mutex.unlock();
}
virtual void burst_write (char *data, unsigned adder, unsigned length)
{
    _bus_mutex.lock();
    wait (length * _cycle_time);
    // copy the data form requestor to memory
    memcpy(_mem+addr, data, length);
    _bus_mutex.unlock();
}
protected:
char* _mem;
sc_time _cycle_time;
sc_mutex _bus_mutex
};
```

Example 7: Simple Bus (1/6)



Example 7: Simple Bus (2/6)



■ Interface

“bus_if.h”

```
#include <systemc.h>
class bus_if: public sc_interface {
public:
    virtual void write(unsigned addr, int data) = 0;
    virtual void read(unsigned addr, int& data) = 0;
};
```

Example 7: Simple Bus (3/6)



Slave – RAM

```
#include "bus_if.h"
class ram:
    public bus_if,
    public sc_module {
public:
    // interface function
    void write(unsigned addr, int data);
    void read(unsigned addr, int& data);

    // constructor
    ram(sc_module_name);
    // destructor
    ~ram();
private:
    // memory contents
    int* pMem;
};
```

“ram.h”

```
#include "ram.h"
ram::ram(sc_module_name nm)
    : sc_module(nm)
{
    pMem = new int[16];
}

ram::~~ram()
{
    delete[] pMem;
}

void ram::write(unsigned addr, int data)
{
    pMem[addr] = data;
}

void ram::read(unsigned addr, int& data)
{
    data = pMem[addr]
}
```

“ram.cpp”

Example 7: Simple Bus (4/6)



Slave – ROM

```
#include "bus_if.h"
class rom:
    public bus_if,
    public sc_module {
public:
    // interface function
    void write(unsigned addr, int data);
    void read(unsigned addr, int& data);

    // constructor
    rom(sc_module_name);
    // destructor
    ~rom();
private:
    // memory contents
    int* pMem;
};
```

“rom.h”

```
#include "rom.h"
rom::rom(sc_module_name nm)
    : sc_module(nm)
{
    pMem = new int[16];
    for(int i=0; i<16; i++) pMem[i] = i;
}

rom::~~ram()
{
    delete[] pMem;
}

void rom::write(unsigned addr, int data)
{
    assert(0);
}

void rom::read(unsigned addr, int& data)
{
    data = pMem[addr];
}
```

“rom.cpp”

Example 7: Simple Bus (5/6)



■ Master

```
#include "bus_if.h"
class cpu: public sc_module {
public:
    // port declaration
    sc_in_clk      clk;
    sc_port<bus_if> mem_port;

    // process declaration
    void cthread_func();

    // constructor
    cpu(sc_module_name);
};
```

“cpu.h”

```
#include "cpu.h"
cpu::cpu(sc_module_name nm)
    : sc_module(nm)
{
    SC_HAS_PROCESS(cpu);
    SC_CTHREAD(cthread_func, clk.pos);
}

void cpu::cthread_func()
{
    while(1)
    {
        int data;
        for(int i=0; i<16; i++) {
            mem_port->read(i, data);
            mem_port->write(i+16, data);
        }
        sc_stop();
    }
}
```

“cpu.cpp”

Example 7: Simple Bus (6/6)



■ Bus

```
#include "bus_if.h"
class bus:
    public bus_if,
    public sc_module {
public:
    // port declaration
    sc_port<bus_if>    rom_port;
    sc_port<bus_if>    ram_port;
    // interface function
    void write(unsigned addr, int data);
    void read(unsigned addr, int& data);

    // constructor
    bus(sc_module_name);
};
```

“bus.h”

```
#include "bus.h"
bus::bus(sc_module_name nm)
    : sc_module(nm)
{
}

void bus::write(unsigned addr, int data)
{
    if(addr < 16)
        rom_port->write(addr, data);
    else
        ram_port->write(addr, data);
}

void bus::read(unsigned addr, int& data)
{
    if(addr < 16)
        rom_port->read(addr, data);
    else
        ram_port->read(addr, data);
}
```

“bus.cpp”

Reference



-
- [1] Chien-Nan Liu, “SystemC Modeling & HW/SW Co-Verification”,
http://www.cs.ccu.edu.tw/~pahsiung/courses/soc/notes/02_Modeling.pdf
 - [2] SystemC Version 2.0 User Guide,
<http://www.cse.iitd.ernet.in/~panda/SYSTEMC/LangDocs/UserGuide20.pdf>