



SystemC Tutorial (I)

Original Slide by 蘇培陞 Alan P. Su

Ver.2 by C. H. Chao, 2006. 9. 21

Ver.3 by Shao-Yi Chien, Feb 27, 2008

Ver.4 by Shao-Yi Chien, Feb 25, 2009


Ver.4.5 by Shao-Yi Chien, March 3, 2017

**Ref: David C. Black and Jack Donovan,
SystemC: From the Ground Up, 2nd Ed.,
Springer, 2009.**



Contents

- ▶ Chapter 0 Introduction
- ▶ Chapter 1 SystemC Overview
- ▶ Chapter 2 C/C++ Basics
- ▶ Chapter 3 Module & Template
- ▶ Chapter 4 Notion of Time
- ▶ Chapter 5 Signal, Port & Binding
- ▶ Chapter 6 Concurrency



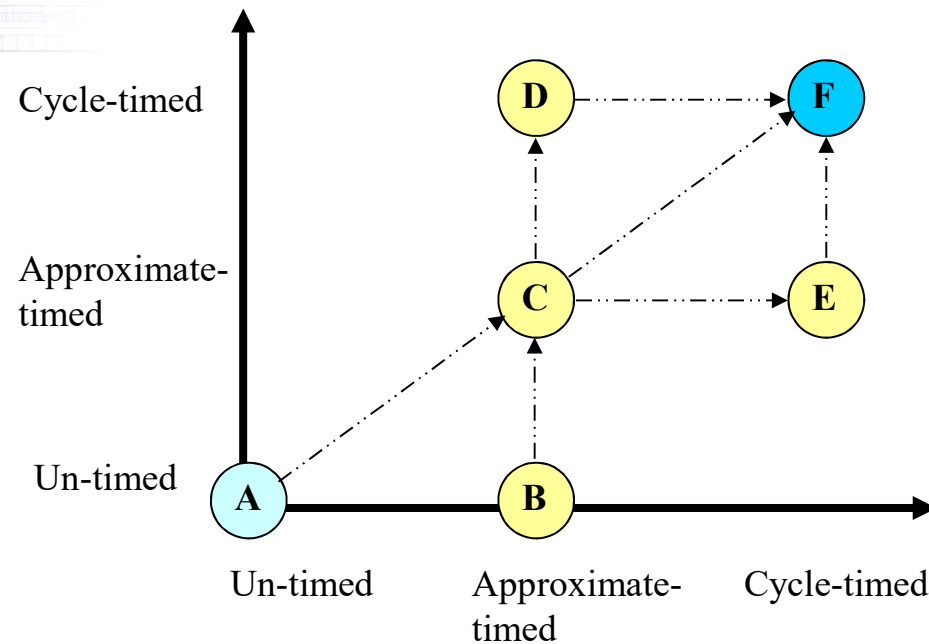
Chapter 0 Introduction

The Concept of Transaction Level
Modeling (TLM)

Different Abstraction Models(1/3)



Communication



- A: Specification model
- B: Component-assembly model
- C: Bus-arbitration model
- D: Bus-functional model
- E: Cycle-accuracy computation model
- F: RTL model
- B,C,D,E are TLMs

Function

Source: L. Cai and D. Gajski, "Transaction Level Modeling: An Overview" CODES+ISSS'03

System Modeling Graph (2/3)

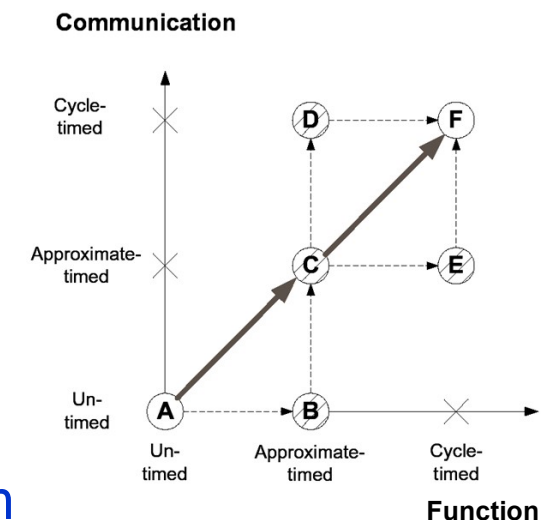


- ▶ X-axis: computation, y-axis: communication
- ▶ Three degrees of time accuracy
 - **Un-timed** computation/communication
 - ▶ pure functionality of the design without any implementation details
 - **Approximate-timed** computation/communication
 - ▶ contains **system-level implementation details**, such as the selected system architecture, the mapping relations between **processes** of the system specification and the **processing elements** of the system architecture
 - ▶ the execution time is usually **estimated** at the system level **without** cycle-accurate RTL/ISS (instruction set simulation) level evaluation
 - **Cycle-timed** computation/communication
 - ▶ contains implementation details at both system level and the RTL/ISS level, such that **cycle-accurate estimation** can be obtained

Different Abstraction Models(2/3)



- ▶ Specification model (A)
 - The functionality of the system is specified in this abstraction level.
- ▶ Component-assembly model (B)
 - The system may be composed of CPU,DSP or other IPs. The system architecturer can estimate the **computational time** without consideration of the communication time. **The number of required processing elements is determined in this level.**
- ▶ Bus arbitration model (C)
 - The data transfer is implemented by the **message-passing channel** without cycle-accuracy, pin-accuracy and specific detailed protocol.
 - The IPs are specified as **master or slave** completely. The **bus arbiter** is required when multiple master IPs exist. The **initial arbitration scheme** is also defined.



Different Abstraction Models(3/3)



► Bus functional model (D)

- The message-passing channels are replaced by **protocol channels** via the procedure of protocol refinement.
- The protocol channels are both **cycle-accurate** and **pin-accurate** with the specific protocol.

► Cycle-accurate computation model (E)

- The **processing elements (PE) or IPs** are **cycle-accurate** and pin-accurate which may be RTL models.
- The **converters or adapters** are required to convert **data transfer** between the **higher** abstraction channel model and **lower** abstraction PE or IP models.

► RTL model (F)

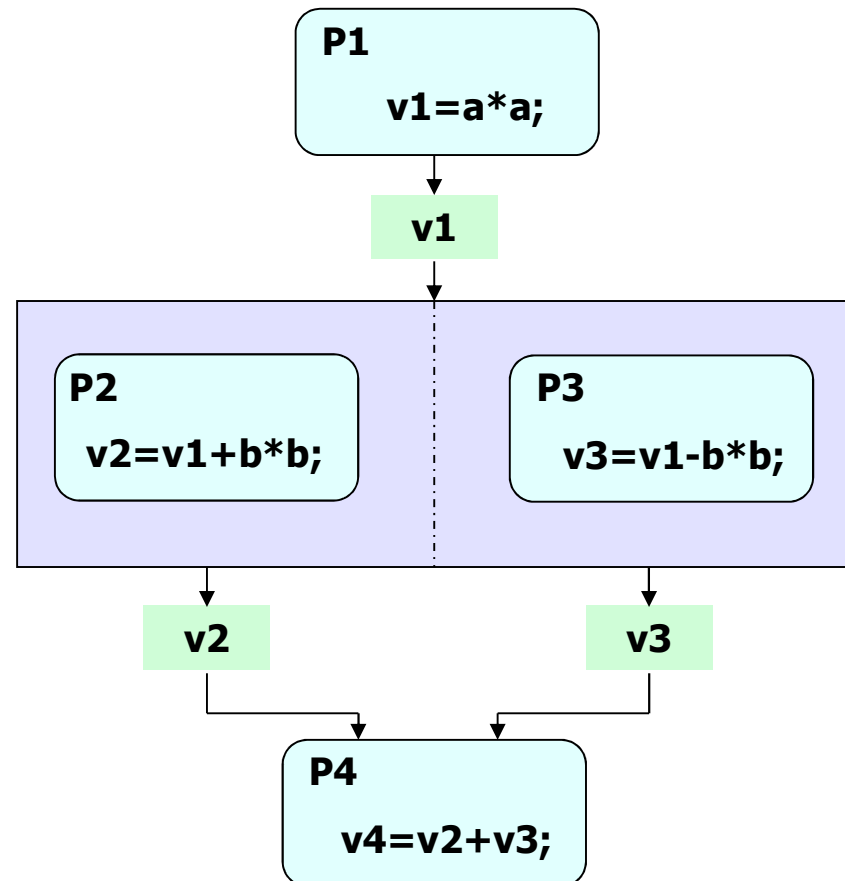
- The overall system architecture is the RTL model which can be synthesized into gate level netlist.
- The synthesizable RTL model can be used for back-end physical IC design and manufacturing procedures.

Specification Model (A)



**P1 ,P2 ,P3 and P4: Process
(function)**

v1, v2 and v3: variables

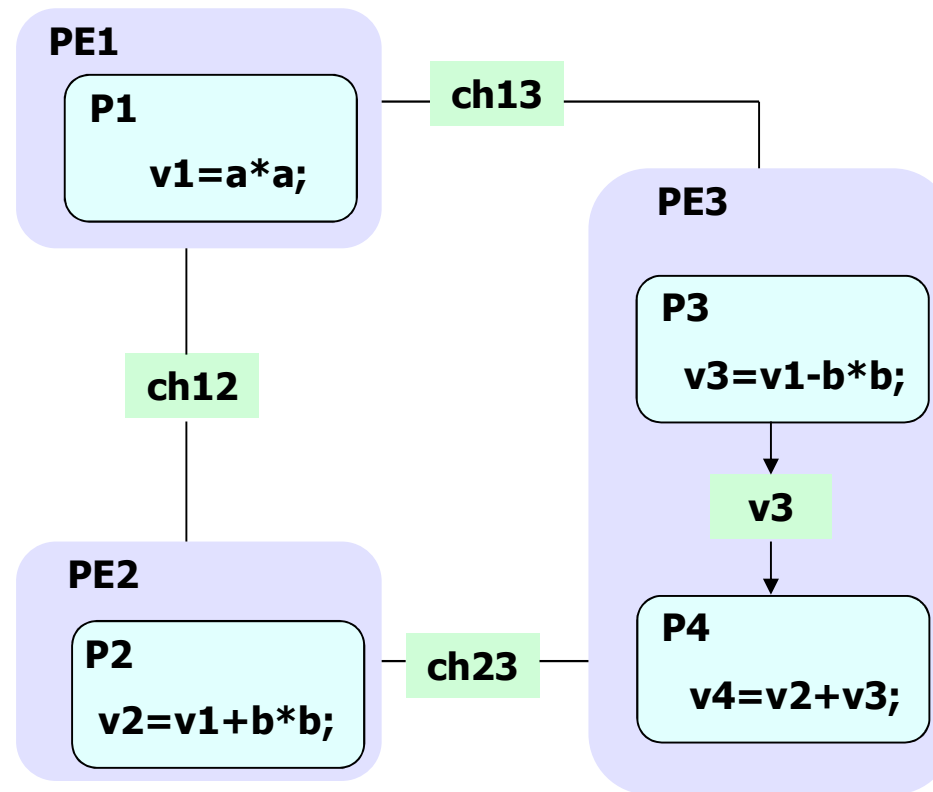
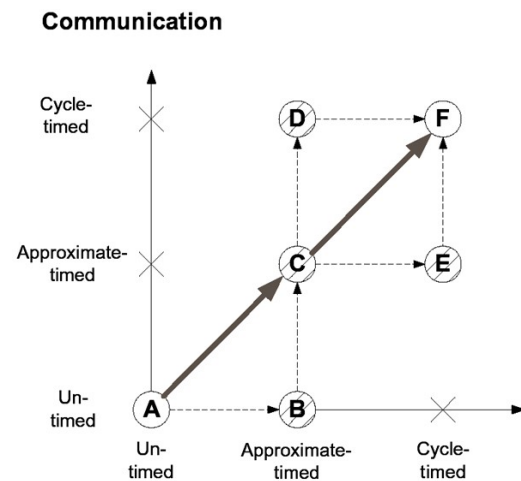


Component-Assembly Model (B)



Ch12, ch13 and ch23:
channels

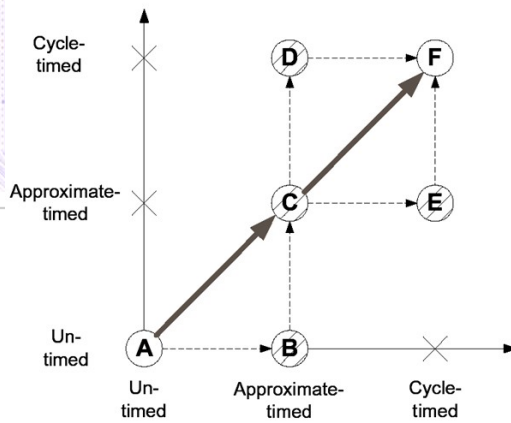
PE1, PE2 and P3:
Processing elements



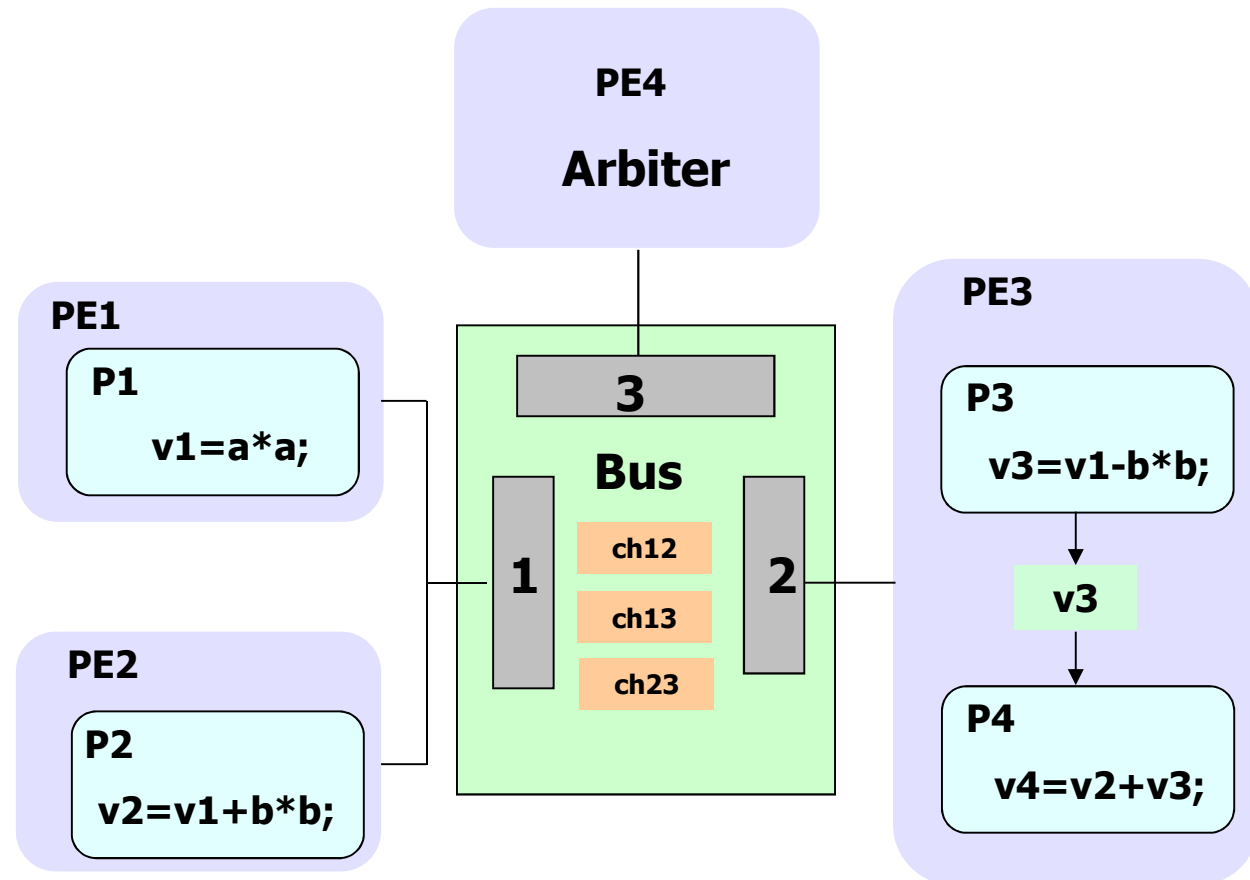
Bus Arbitration Model (C)



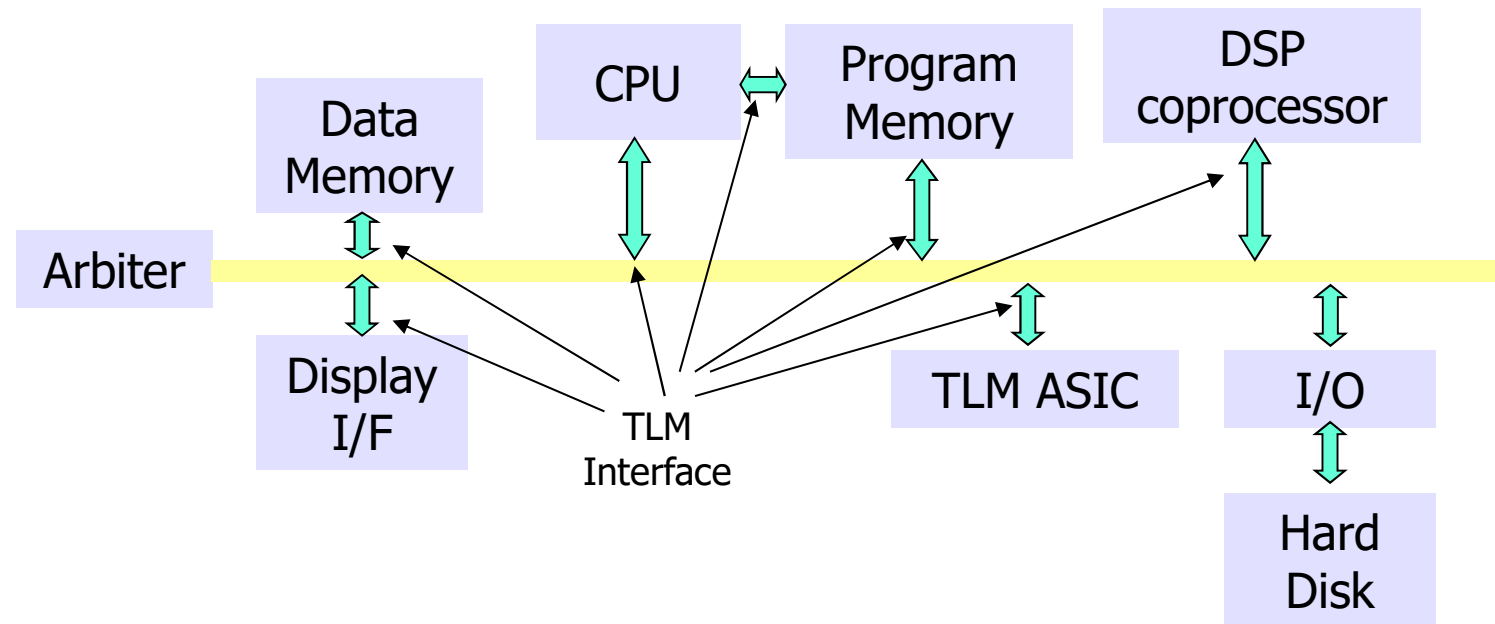
Communication



- 1: Master interface
- 2: Slave interface
- 3: Arbiter interface



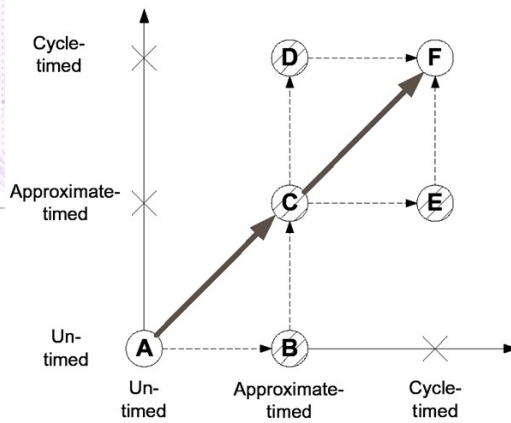
Generic System



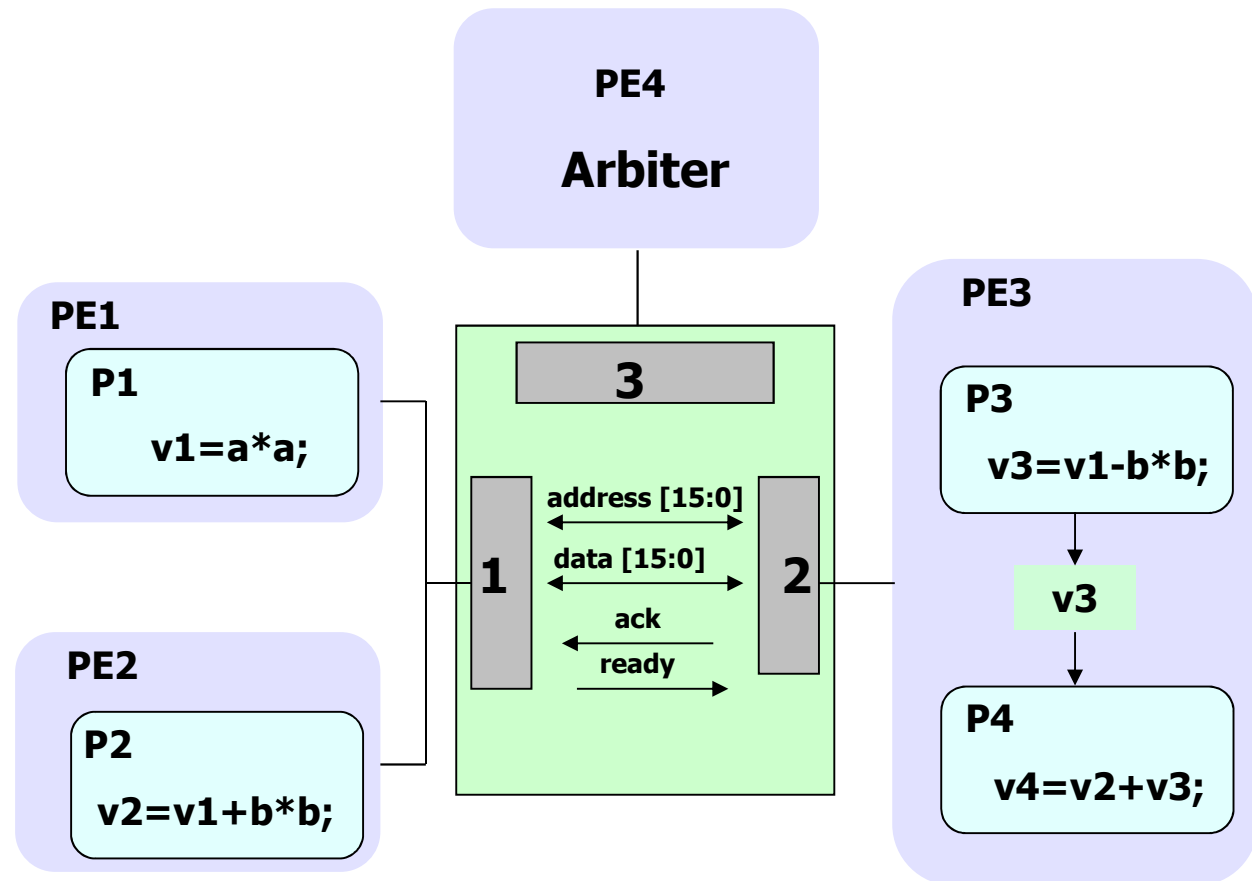
Bus Functional Model (D)



Communication



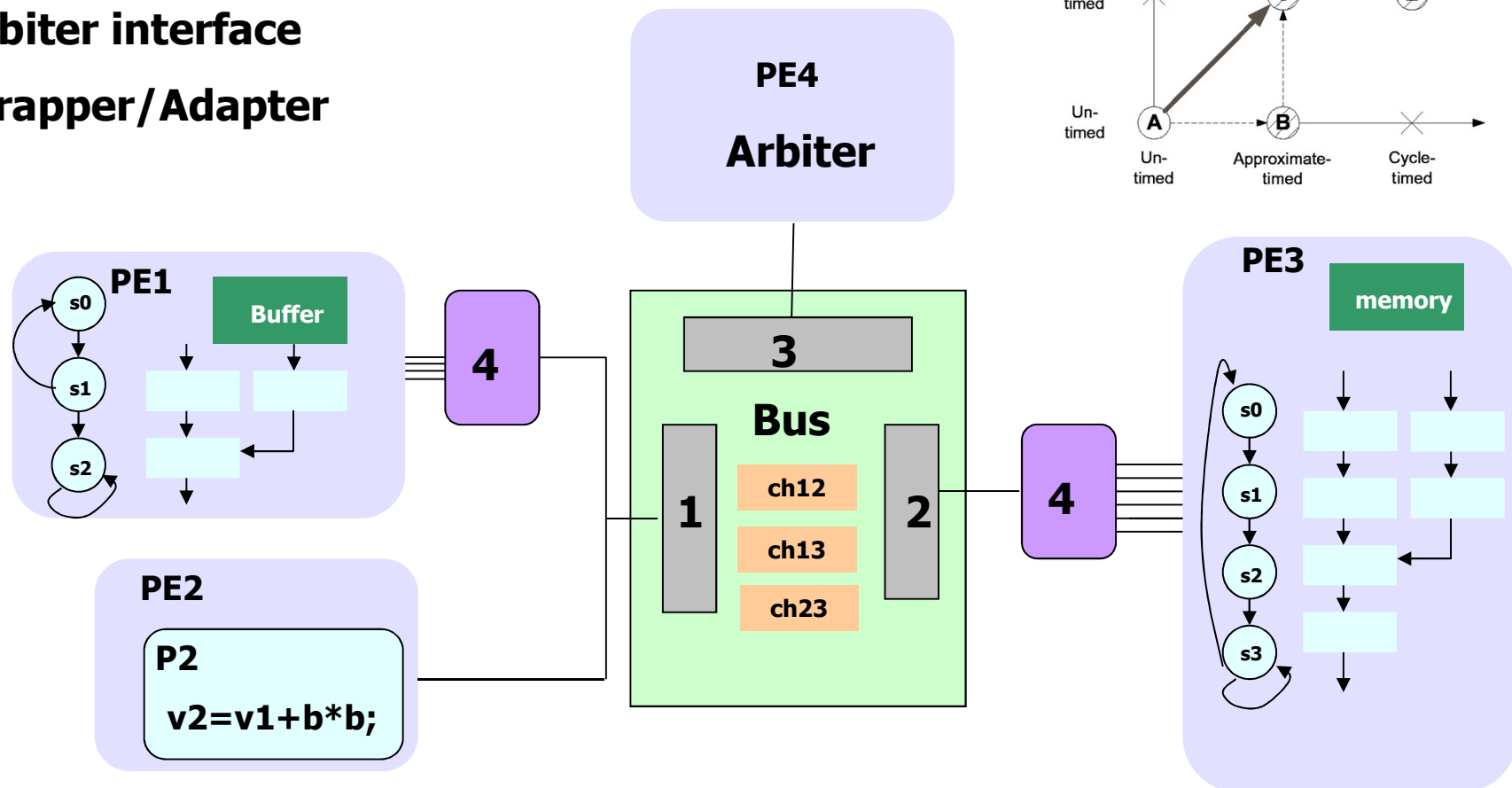
- 1: Master interface
- 2: Slave interface
- 3: Arbiter interface



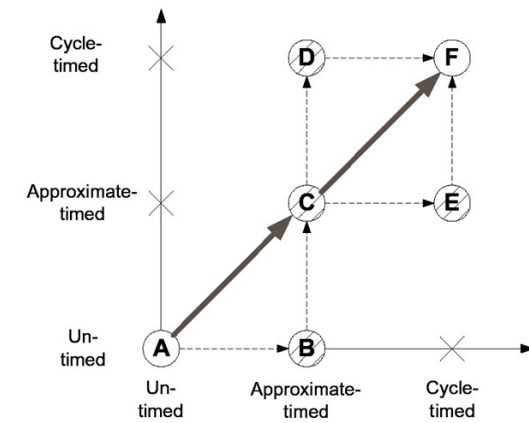
Cycle-Accurate Computation Model (E)



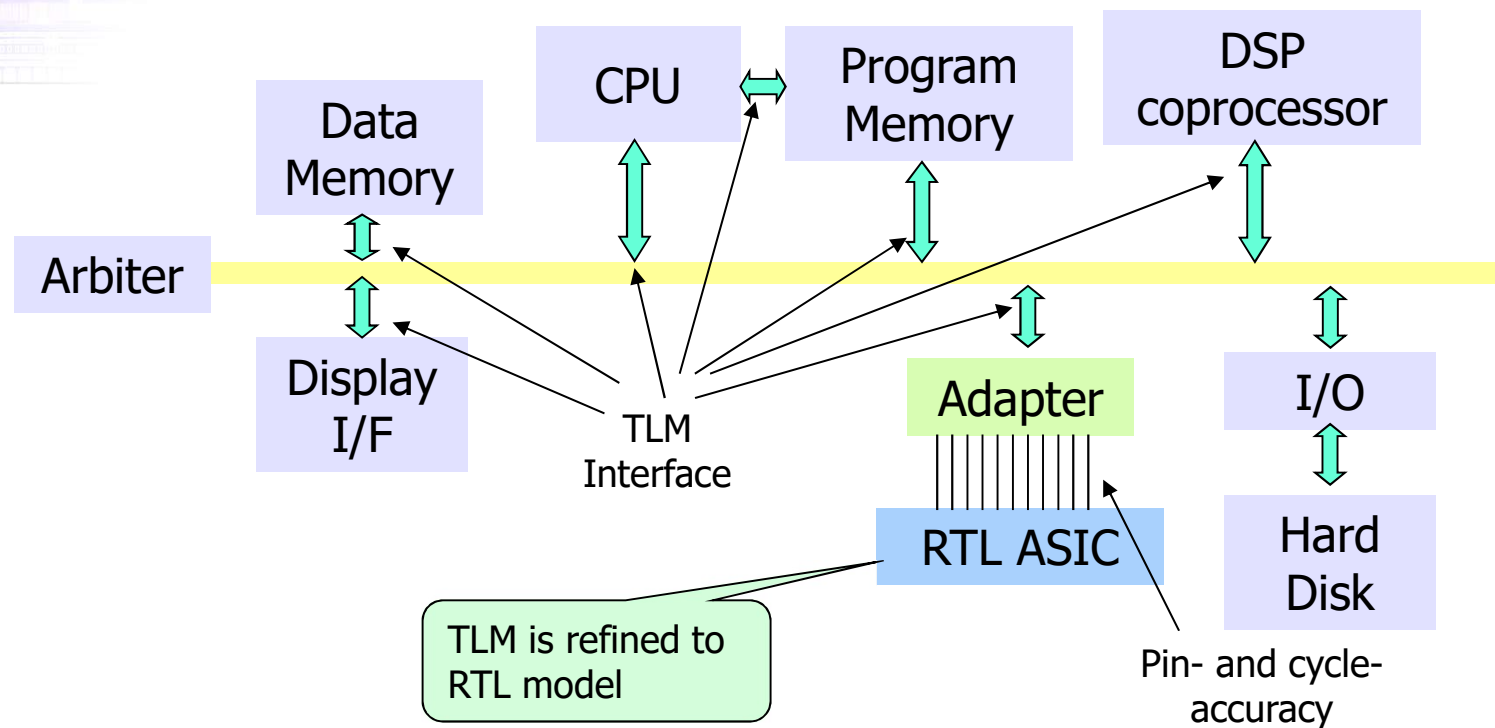
- 1: Master interface
- 2: Slave interface
- 3: Arbiter interface
- 4: Wrapper/Adapter



Communication



Cross Level Modeling with Adapter

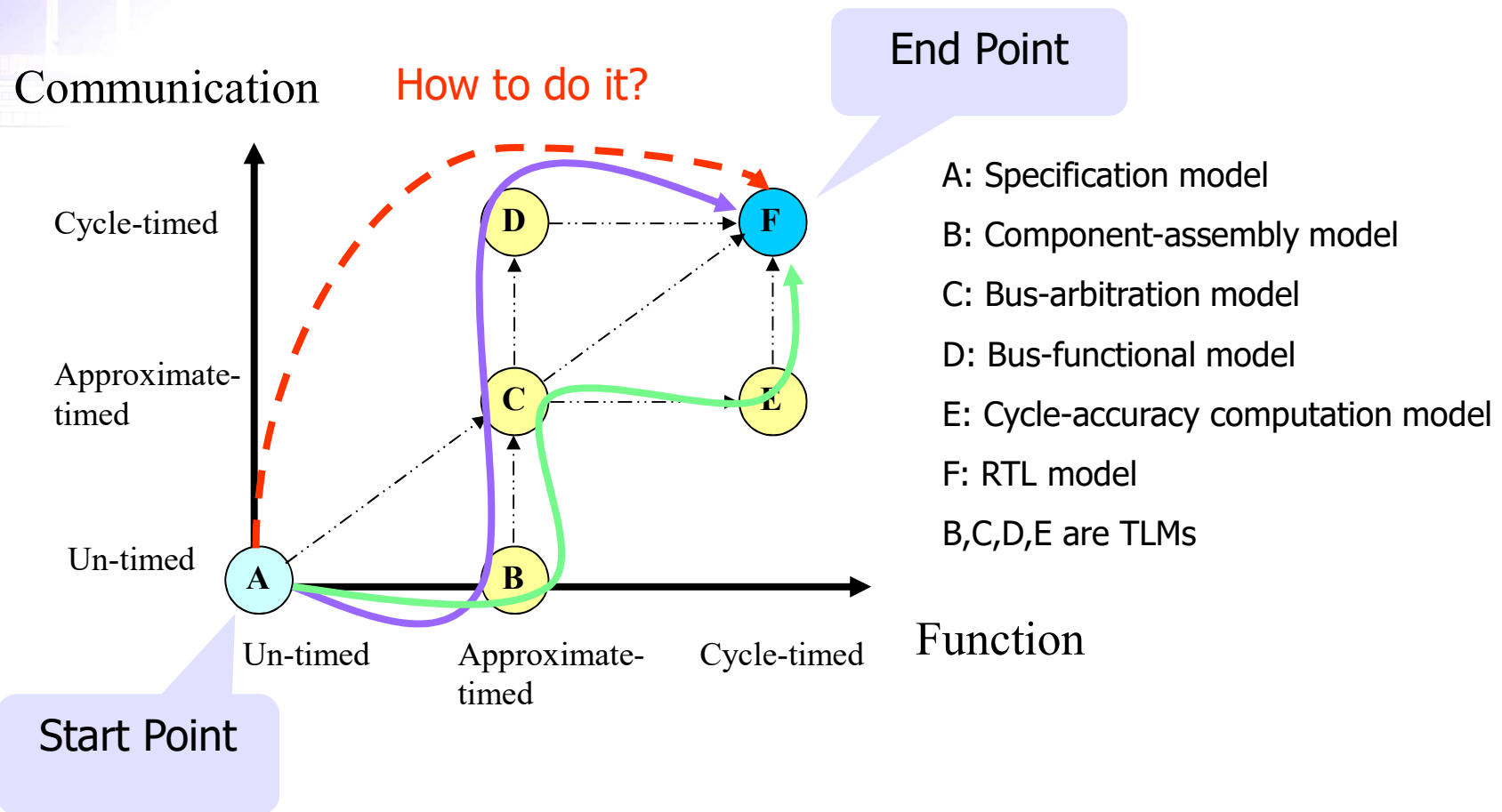


Timing Accuracy of Transaction Level Modeling



Model	Communication	Functionality
Speciation	Un-timed	Un-timed
Component-assembly	Un-timed	Approximate-timed
Bus arbitration	Approximate-timed	Approximate-timed
Bus functional	Cycle-timed	Approximate-timed
Cycle-accurate computation	Approximate-timed	Cycle-timed
RTL	Cycle-timed	Cycle-timed

TLM Design Flow

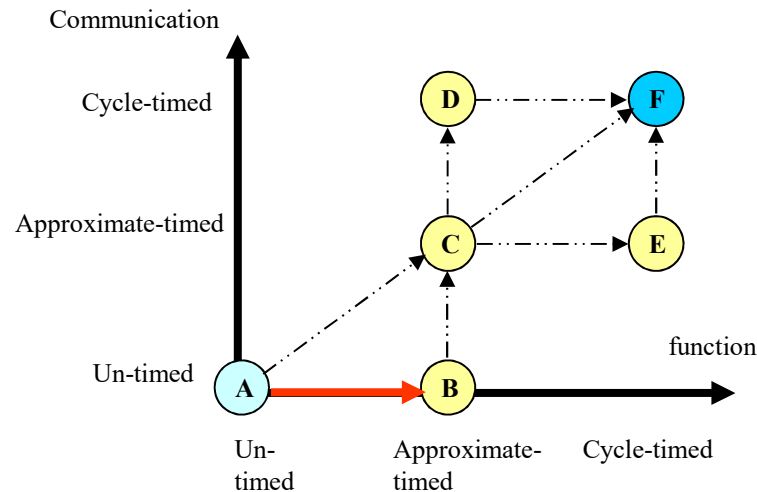


Component Assembly



► Based on the analysis of the algorithm we need to:

- partition the algorithm into Software/Hardware
- select the general purpose processor or the DSP
- design IPs or select IPs from library
- choose RTOS if necessary

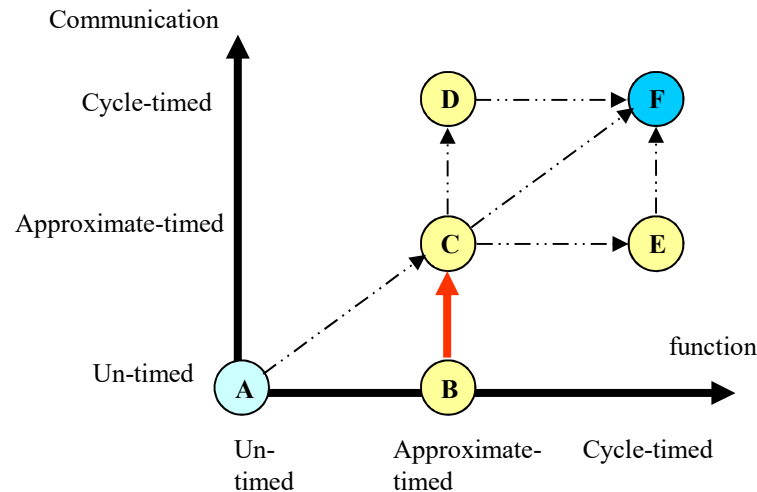


Communication Exploration

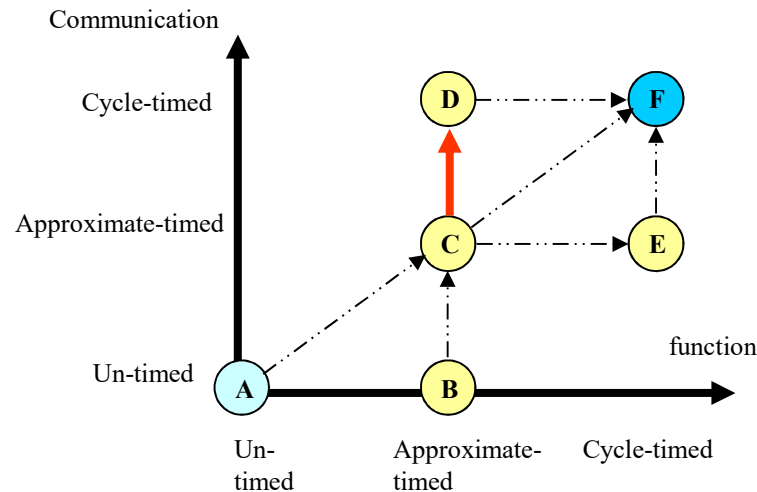


► We need to

- map channels to buses (centralized or back-door)
- assign bus-accessing properties for each IP (master or slave)
- decide the bus arbitration policy

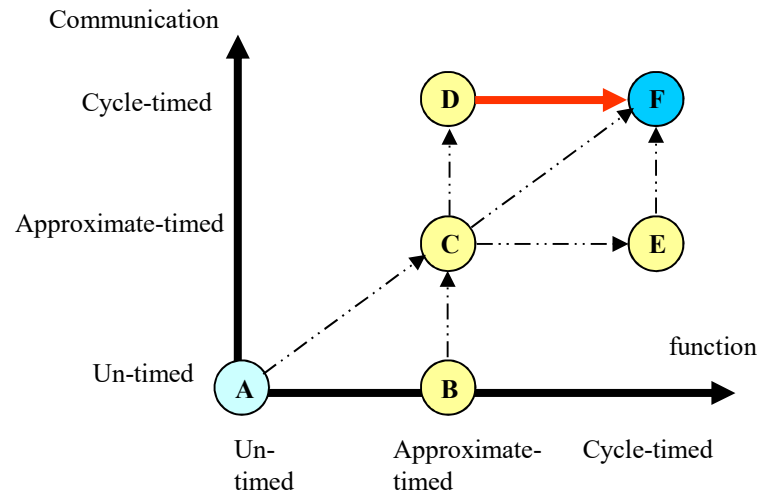


Protocol Refinement (Platform-based)



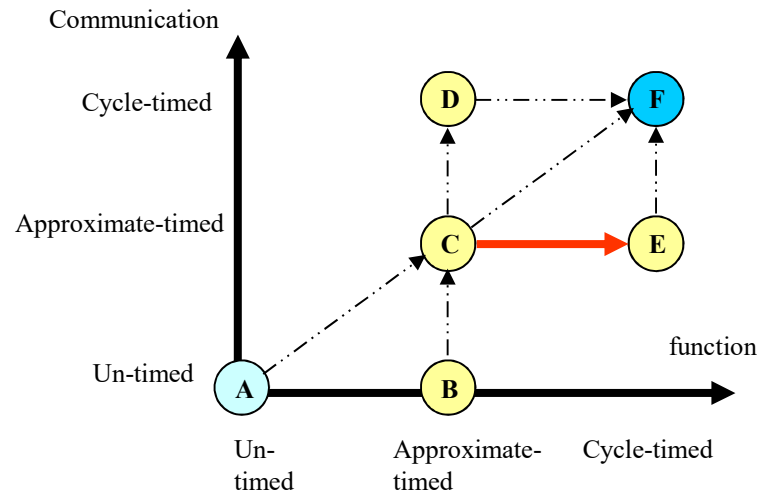
- ▶ We need to determine the pin- and cycle-accurate bus protocols.
- ▶ And the details of the bus control signal are contained.

IP Refinement



- ▶ The IPs are refined to pin- and cycle-accuracy.
- ▶ The embedded software is optimized to achieve high performance.
- ▶ The **wrapper** to transfer the data between IPs and bus are designed.

IP Replacement

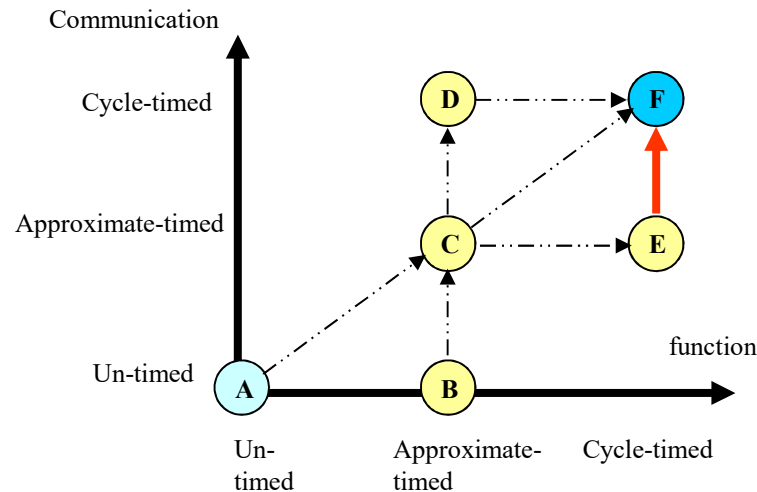


- ▶ Some **important IPs** are modeled with pin- or cycle-accuracy.
- ▶ The cross level **adaptors** are required to bridge the models in different abstraction level.
- ▶ The IPs are replaced or refined one by one.

Communication Refinement



- ▶ We should decide the pin- and cycle-accurate bus protocol.
- ▶ The wrapper to transfer the data between IPs and bus are required.



Other Point of View



- ▶ Un-Timed (UT)
- ▶ Loosely Timed (LT)
- ▶ Approximated Timed (AT)
- ▶ Register Transfer Logic (RTL)
- ▶ Pin and Cycle Accurate (PCA)

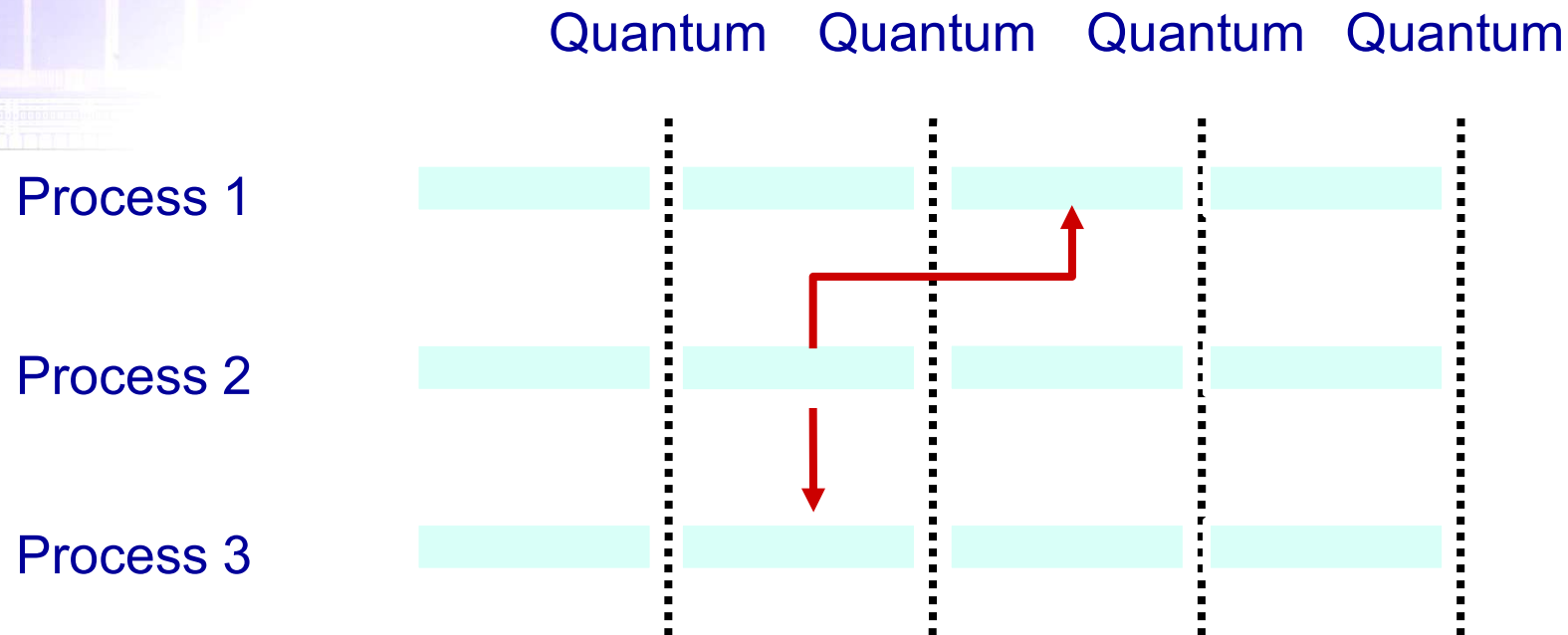
Model Functionality

Model Functionality	RTL	AT	LT	UT
	RTL	BFM	TLM	TLM
	TLM	TLM	TLM	TLM
	TLM	TLM	TLM	TLM
Model Interface	UT	LT	AT	PCA

SAM: System Architecture Model

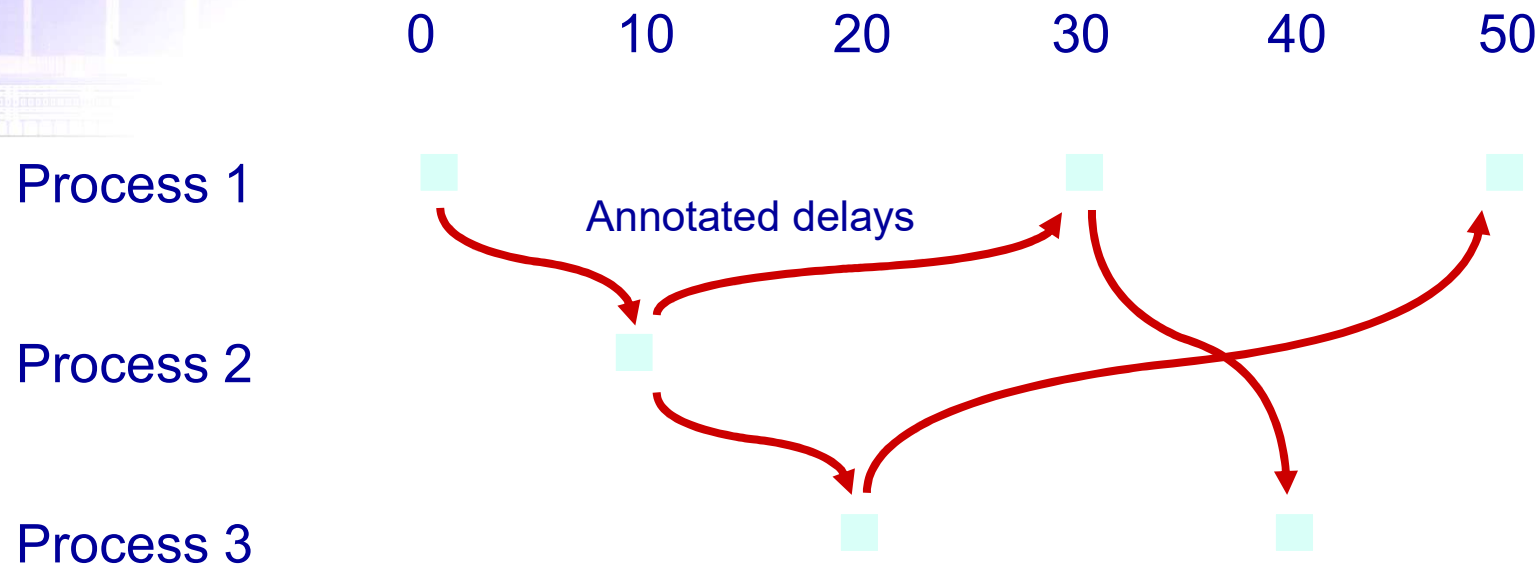
Model Interface

Loosely-timed

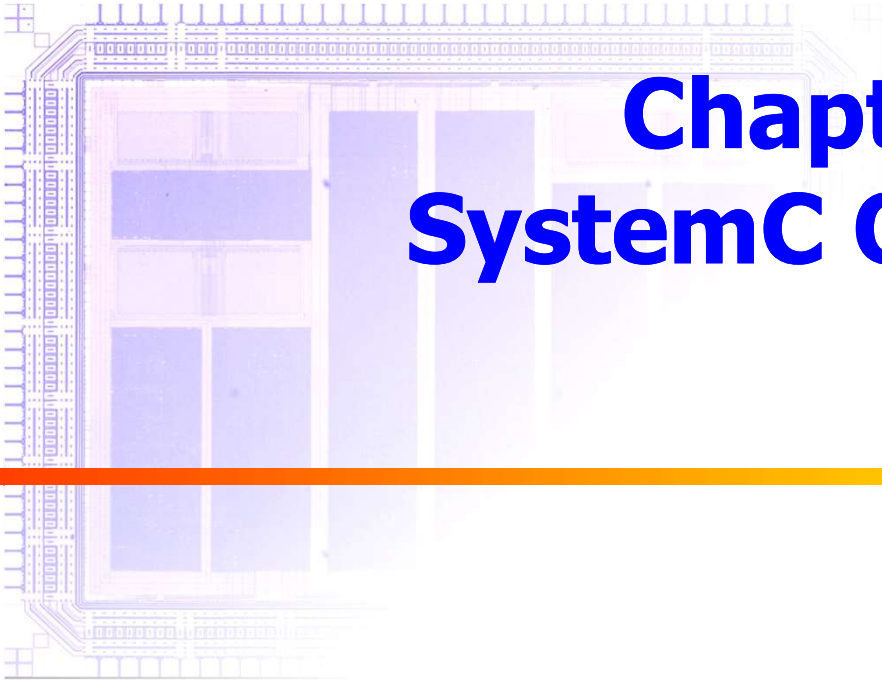


Each process runs ahead up to quantum boundary
sc_time_stamp() advances in multiples of the quantum
Deterministic communication requires explicit synchronization

Approximately-timed



*Each process is synchronized with SystemC scheduler
Delays can be accurate or approximate*



Chapter 1

SystemC Overview

SystemC Overview



- ▶ 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
- ▶ Version 2.1: adding some programming language features and simulation semantics, e.g. `sc_spawn`, `before_end_of_elaboration`, etc.
- ▶ Version 2.2: fix some bugs, match with the IEEE 1666 SystemC Language Reference Manual
- ▶ Version 2.3: match with the IEEE 1666-2011 SystemC Language Reference Manual

New in SystemC 2.3



- ▶ Process Control
- ▶ Stepping and Pausing the Scheduler
 - ▶ `sc_vector`
 - ▶ Integrated with TLM-2.0
 - ▶ SystemC and O/S Threads

SystemC Overview



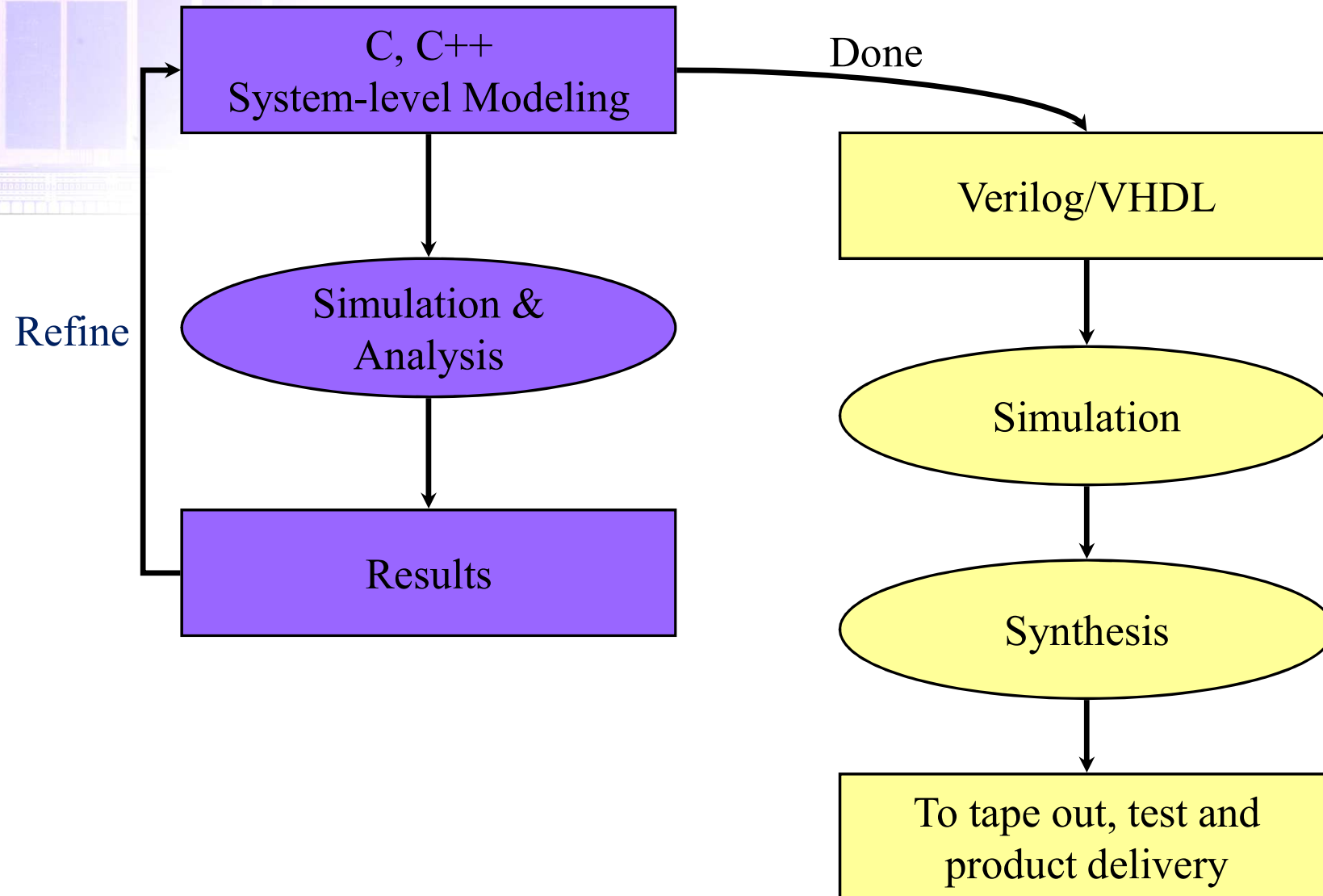
- ▶ Is a C++ class library and a methodology that one can use to effectively create cycle-accurate models of functions, hardware architecture, and interfaces of the SoC and system-level designs.
- ▶ One can use SystemC and standard C++ development tools to create a system-level model, quickly simulate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

SystemC Highlights

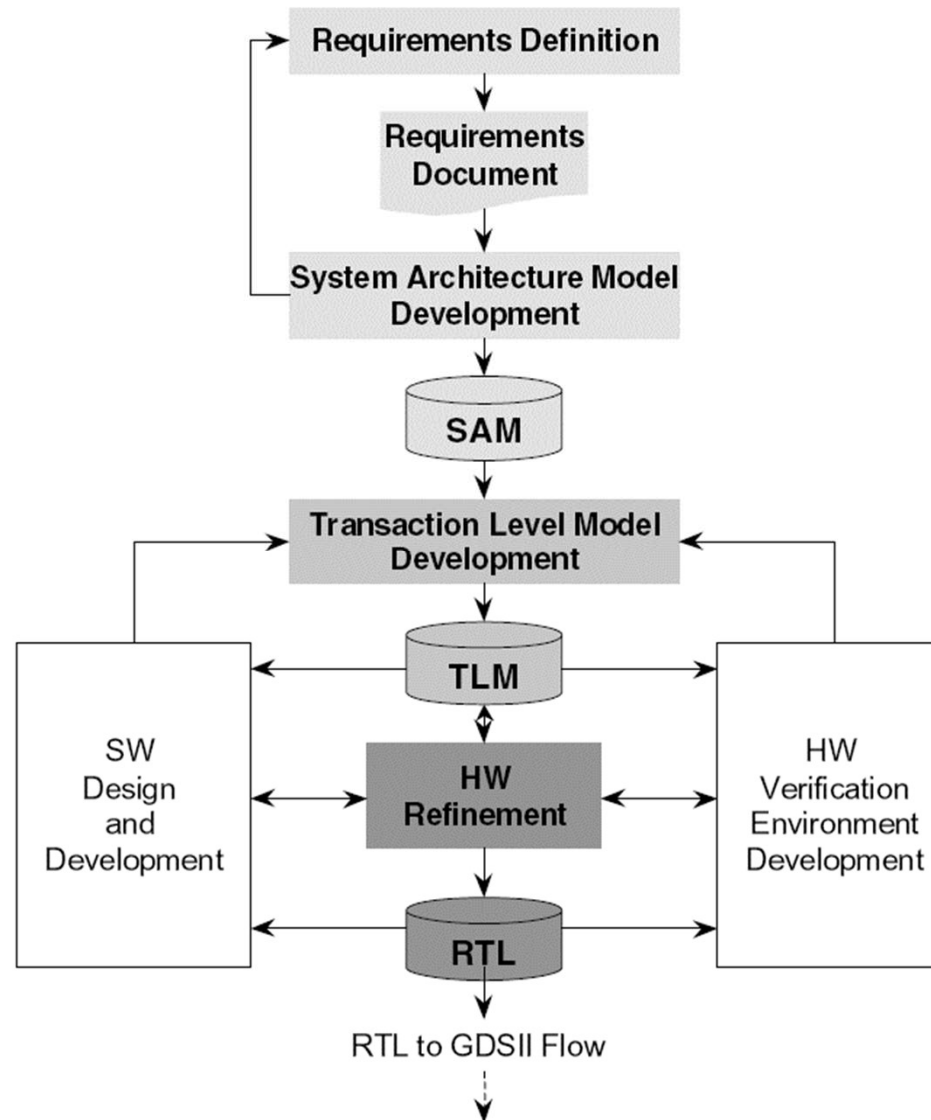


- ▶ Modules: component
- ▶ Processes: functions, SC_THREAD & SC_METHOD
- ▶ Ports: I/O ports
- ▶ Signals: wires
- ▶ Rich set of port and signal types
- ▶ Rich set of data types
- ▶ Clocks
- ▶ Cycle-based simulation: ultra light-weight and fast
- ▶ Multiple abstraction levels
- ▶ Communication protocols: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

Current System Design Methodology



TLM Based Flow



SystemC 2.0 Language Architecture



臺灣大學

Methodology-Specific Libraries

Master/Slave Library, etc.

Layered Libraries

Verification Library
Static Dataflow, etc.

Primitive Channels

Signal, Mutex, Semaphore, FIFO, etc.

Core Language

Modules
Ports
Processes
Interfaces
Channels
Events

Event-Driven Simulation

Data Types

4-valued Logic Type
4-valued Logic Vectors
Bits and Bit Vectors
Arbitrary Precision Integers
Fixed-Point Types
C++ User-Defined Types

C++ Language Standard

SystemC 2.1 Language Architecture



Methodology-Specific Libraries

Master/Slave Library, etc

Layered Libraries

Verification Library,
TLM Library, etc.

Primitive Channels

Signal, Mutex, Semaphore, FIFO, etc.

Core Language

Modules
Ports
Interfaces
Channels

Data Types

4-valued Logic Type
4-valued Logic Vectors
Bits and Bit Vectors
Arbitrary Precision Integers
Fixed-Point Types

Event-Driven Simulation

Events, Processes

C++ Language Standards

SystemC Language Architecture

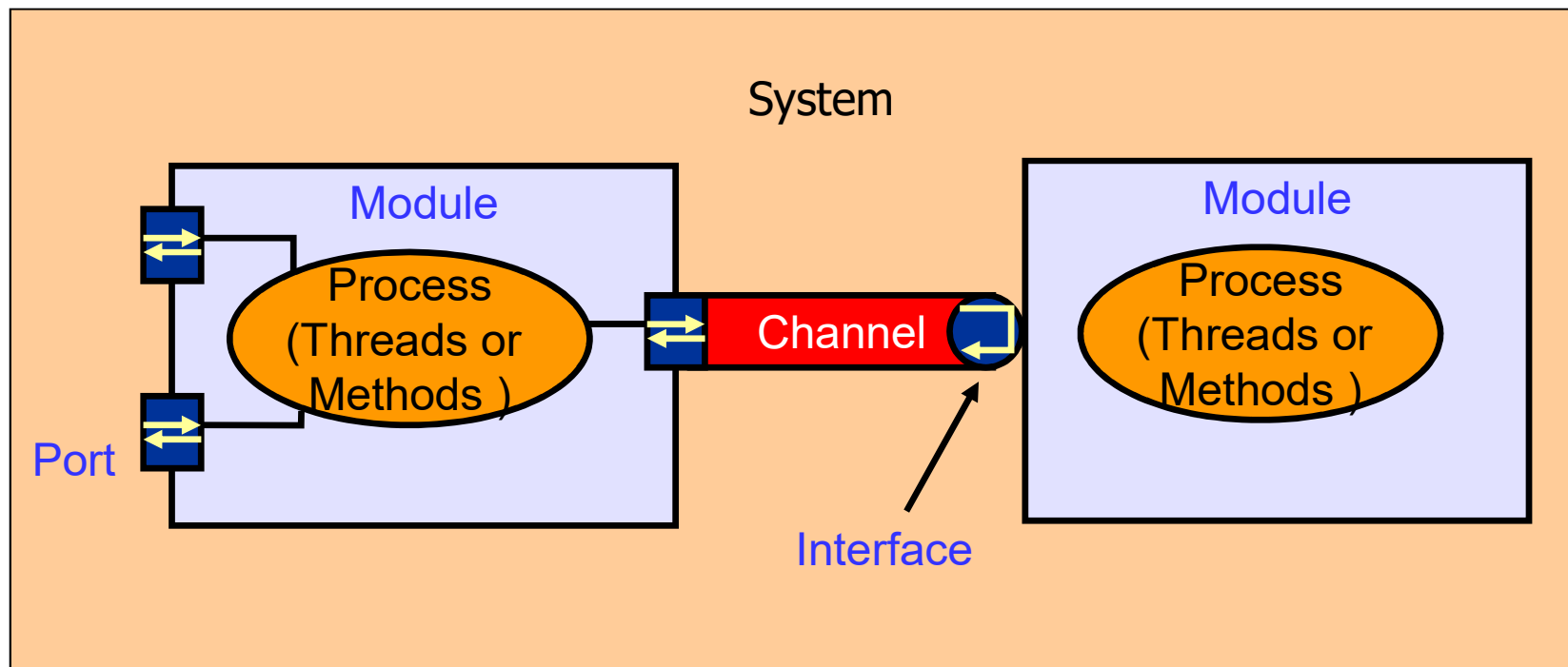
SystemC

User libraries		SCV		Other IP	
Predefined Primitive Channels: Mutexs, FIFOs, & Signals					
Simulation Kernel	Threads & Methods		Channels & Interfaces		Data types: Logic, Integers, Fixed point
	Events, Sensitivity & Notifications		Modules & Hierarchy		
C++					STL

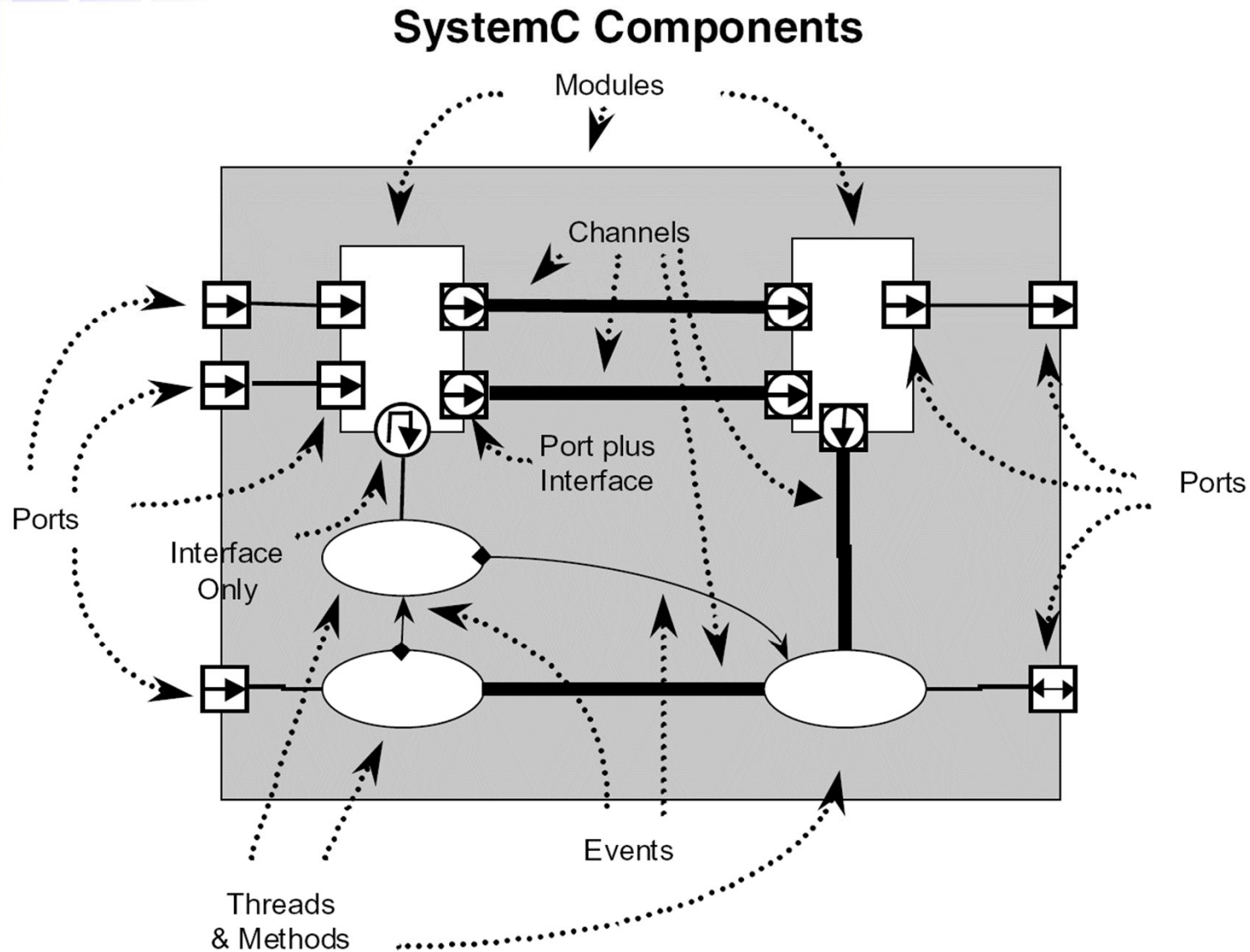
Basic Structure



- ▶ Basic Structure of SystemC model
 - Module
 - Port, interface and channel
 - Process



SystemC Component



SystemC & C++



- ▶ SystemC is a set of C++ class and definitions 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 and such.
- ▶ However you need to follow SystemC methodology closely to make sure the simulation executes correctly.

SystemC & HDL



- ▶ SystemC is a Hardware Description Language (HDL) from system-level down to gate level.
- ▶ Modules written in traditional HDLs like Verilog and VHDL can be translated into SystemC, but not vice versa. Reason: Verilog and VHDL do not support transaction-level.
- ▶ System-Verilog is Verilog plus assertion, which is an idea borrowed from programming languages. And SystemC supports assertion as well through the C++ syntax and semantics.

SystemVerilog vs. SystemC



- ▶ SystemVerilog is Verilog plus verification (assertion).
- ▶ Actually the above statement is not fair but it is the truth now.
- ▶ SystemVerilog and SystemC work together to complete the design platform from system-level to gate-level.
- ▶ SystemC deals with whatever above RTL.
- ▶ SystemVerilog deals with RTL and below.

SystemC Myth I



電子工程專輯

新聞和趨勢

欲列印此文章，請從您的瀏覽器下拉式選項中選擇“檔案”後再選“列印”。

走出自我 軟硬體工程師需要共通語言

上網時間：2005年07月18日

系統單晶片(SoC)之設計面臨的一個重要問題在於軟體與硬體設計師們各自生活在自己的世界裡。這是日本名古屋大學(Nagoya University)的資訊教授Hiroaki Takada日前在MPSoC (Multi-Processor SoC)研討會上所發表的看法。

「用一種語言來描述硬體和軟體能夠改進設計生產率。」他表示。但其所指並非以一概全，不是只要一種產業解決方案的意思。他解釋，「我不認為SystemC適合軟體工程師。至少，我不喜歡它。」

SystemC Myth II



- ▶ It is a language to unify the design environment, SW and HW. A unified design environment.
- ▶ Well, this is a dream in the academy. In industry, this is a long way to go and as of today, SystemC is not the answer. Notice, SystemC is an HDL, it itself does not support software performance measure mechanism.
- ▶ Will the day that an unified design language be realized? We just don't know. But people are talking about UML, the Unified Modeling Language.

SystemC **does not**
model software.

It is an HDL.

SystemC **does not**
run faster, higher
abstraction level
does.

System-Level Language

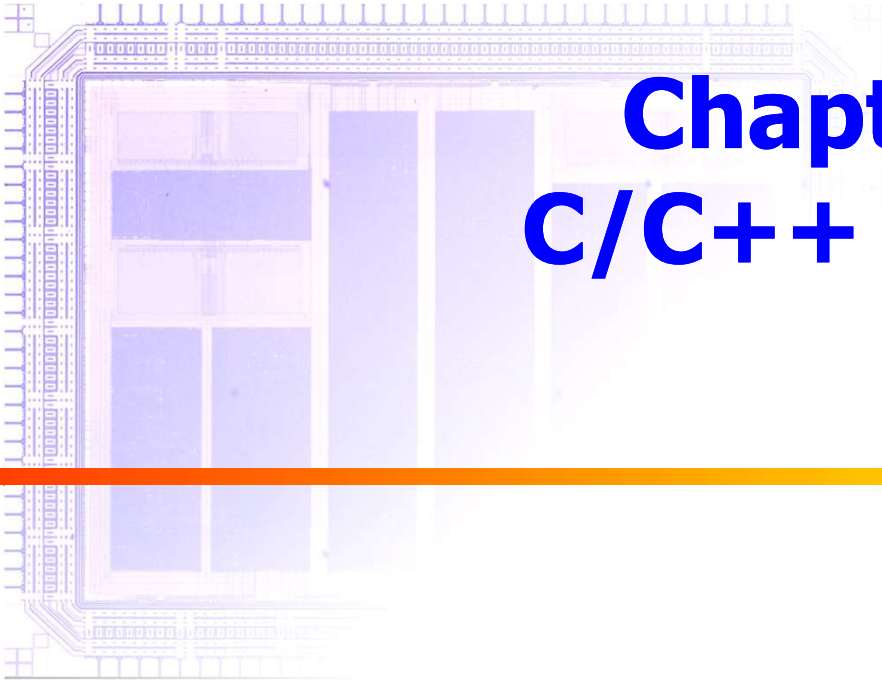


- ▶ To be categorized as a system-level language, the simulation SPEED is the key.
- ▶ The simulation speed should take no 1,000 times slower than the real HW. In another word, 1 second of HW execution time equals 16 minutes and 40 seconds simulation time
- ▶ To achieve this kind of performance, the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

UML



- ▶ UML is a meta-language to model all levels of abstraction.
- ▶ However, it is currently used as a adaptor to link tools in different languages.
- ▶ Mentor Graphics xtUML in Nucleus.
- ▶ Cadence funded UC Berkeley research: Metropolis.
- ▶ XML



Chapter 2

C/C++ Basics

Program Structure



```
main (int argc , char* argv)
{
    int i, j, k;

    scanf("Input i = %d,", i);
    scanf(" j = %d", j);
    k = sum (i, j);
    printf("Output i + j = %d\n", k);

    return 0;
}

int sum(int l, m)
{
    return l + m;
}
```


Data Types

- ▶ int: integer
- ▶ long: double word integer
- ▶ short: 2-byte integer
- ▶ float: single precision floating point
- ▶ double: double precision floating point
- ▶ char: character
- ▶ *: pointer
 - e.g. int* pointer of int, char* pointer of char

Operators

▶ Arithmetic:

$+$, $-$, $*$, $/$: add, subtract, multiply, divide

$\%$: modular, the remainder of a division: $9 \% 2 = 1$

▶ Relational

$>$, $>=$, $<$, $<=$, $==$ (equal to), $!=$ (not equal to)

▶ Logical

$\&\&$ (and), $\|\|$ (or)

▶ Bitwise

$\&$ (AND), $\|$ (OR), \wedge (XOR),

$<<$ (left shift), $>>$ (right shift)

\sim (1's compliment)

Flow Control Statements



► Condition: $a == b$, $a != b$, $c \&\& d$, $e || f$, etc

► if (condition) {

...

} else if (condition) {

...

} else {

...

}

► for ($i = 0$; $i < 10$, $i++$) { ... }

Flow Control Statements Cont.

- ▶ `do { ... } while (condition);`
- ▶ `break`: break out of the for or (do) while loop
- ▶ `continue`: skip to the next for or (do) while loop
- ▶ `switch (c) {`
 - `case '0':`
`printf("c is 0\n");`
 - `case '1':`
`printf("c is 1\n");`
 - `default:`
`printf("c is not 0 nor 1\n");``}`

C++ Basics



- ▶ Class: an object, contains data and function members, must include constructor and destructor

```
class cup {  
    float volume;           // data member  
    cup(float vol);         // constructor  
    ~cup();                 // destructor  
    fill(float vol);        // function member  
    drink(float vol);  
};
```



Chapter 3

Module and Template

SC_MODULE

Starting Point: sc_main



▶ C/C++

```
int main(int argc, char* argv[])
{
    BODY_OF_PROGRAM
    return EXIT_CODE; //Zero indicates success
}
```

▶ SystemC

```
int sc_main(int argc, char* argv[])
{
    ELABORATION
    sc_start(); //← Simulation begins & ends in this function
    [POST-PROCESSING]
    return EXIT_CODE; //Zero indicates success
}
```

Basic Unit of Design: SC_MODULE



- ▶ A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity
- ▶ Syntax

```
#include <systemc.h>
SC_MODULE (module_name) {
    MODULE_BODY
};
```
- ▶ SC_MODULE is a simple cpp macro

```
#define SC_MODULE(module_name) \
struct module_name: public sc_module
```


MODULE BODY



- ▶ Ports
- ▶ Member channel instances
- ▶ Member data instances
- ▶ Member module instances (sub-designs)
- ▶ Constructor
- ▶ Destructor
- ▶ Process member functions (processes)
- ▶ Helper functions

Constructor: SC_CTOR



```
SC_CTOR(module_name)
```

```
: Initialization
```

```
{
```

```
  Subdesign_Allocation
```

```
  Subdesign_Connectivity
```

```
  Process_Registration
```

```
  Miscellaneous_Setup
```

```
}
```

► SystemC process

- void *PROCESS_NAME*(void)

Registering the Simple Process: SC_THREAD



▶ SC_THREAD: like *initial* in Verilog

▶ In simple_process_ex.h

```
#include <systemc.h>
SC_MODULE(simple_process_ex) {
    SC_CTOR(simple_process_ex) {
        SC_THREAD(my_thread_process);
    }
    void my_thread_process(void);
};
```

Implementation



► In simple_process_ex.cpp

```
#include "simple_process_ex.h"
void simple_process_ex::my_thread_process(void) {
    std::cout << "my_thread_process executed within "
                << name()
                << std::endl;
}
```

Main Function

► In main.cpp

```
#include "simple_process_ex.h"
int sc_main(int argc, char* argv[]) {
    simple_process_ex my_instance("my_instance");
    sc_start();
    return 0;
}
```

Alternative Constructors: SC_HAS_PROCESS



- ▶ Can transfer more than instance name to configure the module
 - My_memory instance("instance", 1024);

```
//FILE: module_name.h
SC_MODULE(module_name) {
    SC_HAS_PROCESS(module_name);
    module_name (sc_module_name instname[, other_args...]);
};
```

```
//FILE: module_name.cpp
module_name::module_name(
    sc_module_name instname[, other_args...])
: sc_module(instname)
[, other_initializers]
{
    CONSTRUCTOR_BODY
}
```

Alternative Constructors: SC_HAS_PROCESS



清華大學

```
//FILE: module_name.h
SC_MODULE(module_name) {
    SC_HAS_PROCESS(module_name);
    module_name(sc_module_name instname[, other_args...])
    : sc_module(instname)
    [, other_initializers]
    {
        CONSTRUCTOR_BODY
    }
};
```



Chapter 4

Notion of Time

sc_time



► Unit

- SC_SEC //seconds
- SC_MS //milliseconds
- SC_US //microseconds
- SC_NS //nanoseconds
- SC_PS //picoseconds
- SC_FS //femtoseconds

► Syntax

- `sc_time name...;`
- `sc_time name(magnitude, timeunits)...;`

► Examples

- `sc_time t_PERIOD(5, SC_NS);`
- `sc_start(60.0, SC_SEC);`

wait(sc_time)



► Example

```
void simple_process_ex::my_thread_process (void) {  
    wait (10, SC_NS);  
    std::cout<< "Now at "<< sc_time_stamp() << std::endl;  
    sc_time t_DELAY(2, SC_MS); // keyboard debounce time  
    t_DELAY *= 2;  
    std::cout<< "Delaying "<< t_DELAY<< std::endl;  
    wait(t_DELAY);  
    std::cout << "Now at " << sc_time_stamp()  
                << std::endl;  
}
```

```
% ./run_example  
Now at 10 ns  
Delaying 4 ms  
Now at 4000010 ns
```

Other Time Related Functions



- ▶ `double sc_simulation_time()`
- ▶ `sc_set_time_resolution(value, tunit)`
- ▶ `sc_set_default_time_unit(value, tunit)`



Chapter 5

Port, Signal and Binding

Ports

- ▶ `sc_in<data_type>;` — input port
- ▶ `sc_out<data_type>;` — output port
- ▶ `sc_inout<data_type>;` — input/output port
- ▶ If `data_type` is a type with size declaration, a space is needed before the closing bracket
`sc_in<sc_uint<10> >;`

Fast Port Binding



- ▶ While using HW port-to-port binding is not intuitive at system-level and slow. Starting 2.1, SC_EXPORT is supported for fast port binding.
- ▶ Generally used at the inner-most, lowest level models to replace SC_PORT.
- ▶ The purpose is to simplify the port binding between layers.

Signal

- ▶ Is a un-directional wire
- ▶ The flow of data is determined by the ports a signal connects to

▶ Example:

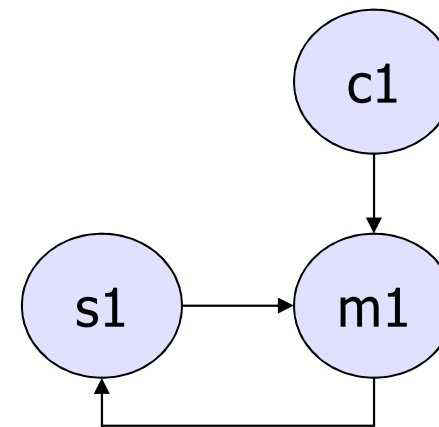
```
sc_signal<sc_uint<32> >;
```

Port & Signal Connection



► Named Mapping

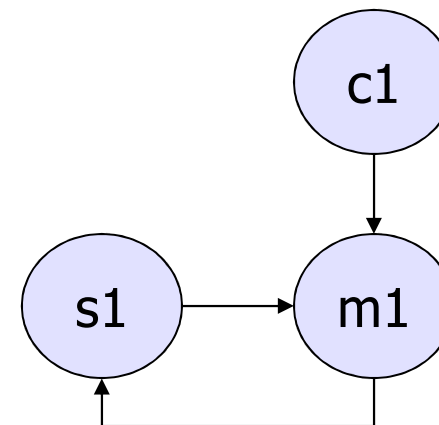
```
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"
SC_MODULE(filter) {
    sample *s1;
    coeff *c1;
    mult *m1;
    sc_signal<sc_uint<32> > q, s, c;
    SC_CTOR(filter) {
        s1 = new sample("s1");
        s1->din ( q );
        s1->dout ( s );
        c1 = new coeff("c1");
        c1->out ( c );
        m1 = new mult("m1");
        m1->a ( s );
        m1->b ( c );
        m1->q ( q );
    }
};
```



Port & Signal Connection Cont.

► Positional Mapping

```
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"
SC_MODULE(filter) {
    sample *s1;
    coeff *c1;
    mult *m1;
    sc_signal<sc_uint<32> > q, s, c;
    SC_CTOR(filter) {
        s1 = new sample("s1");
        (*s1) ( q,s );
        c1 = new coeff("c1");
        (*c1) ( c );
        m1 = new mult("m1");
        (*m1) ( s, c, q );
    }
};
```



Process First Look



- ▶ Processes are the basic unit of execution within SystemC. Processes are called to emulate the behavior of the target device or system.
- ▶ The real work of a module is performed in processes.
- ▶ Processes are functions that are identified to the SystemC kernel and called/activated whenever signals these processes are sensitive to.
- ▶ These statements are executed sequentially until the end of the process, or being suspended by a wait() statement.
- ▶ **SC_METHOD**, SC_THREAD SC_CTHREAD

Module Example



```
#include "systemc.h"
```

```
SC_MODULE(timer) {
```

```
    sc_inout<bool> start;           // ports
```

```
    sc_out<bool> timeout;
```

```
    sc_in<bool> clock;
```

```
    int count;                     // data and function members
```

```
    void runtimer();
```

```
    SC_CTOR(timer) {               // constructor
```

```
        SC_THREAD(runtimer);
```

```
        sensitive_pos << clock;    // sensitivity list
```

```
        sensitive << start;
```

```
        count = 0;
```

```
    }
```

```
};                                  // do not forget the final semi-column
```

Clock

► Synopsis

```
class sc_clock :          public sc_signal_in_if<bool>,
                          public sc_module
{
public:
    sc_clock();
    explicit sc_clock( sc_module_name name_ );
    sc_clock( sc_module_name name_,
              const sc_time& period_,
              double duty_cycle_ = 0.5,
              const sc_time& start_time_ = SC_ZERO_TIME,
              bool pos_edge_first = true);

    ...
};
```

► Example:

```
sc_clock clock1("clock1", 20, 0.50, 2, true);
```

Clock – Difference in 2.0 and 2.1

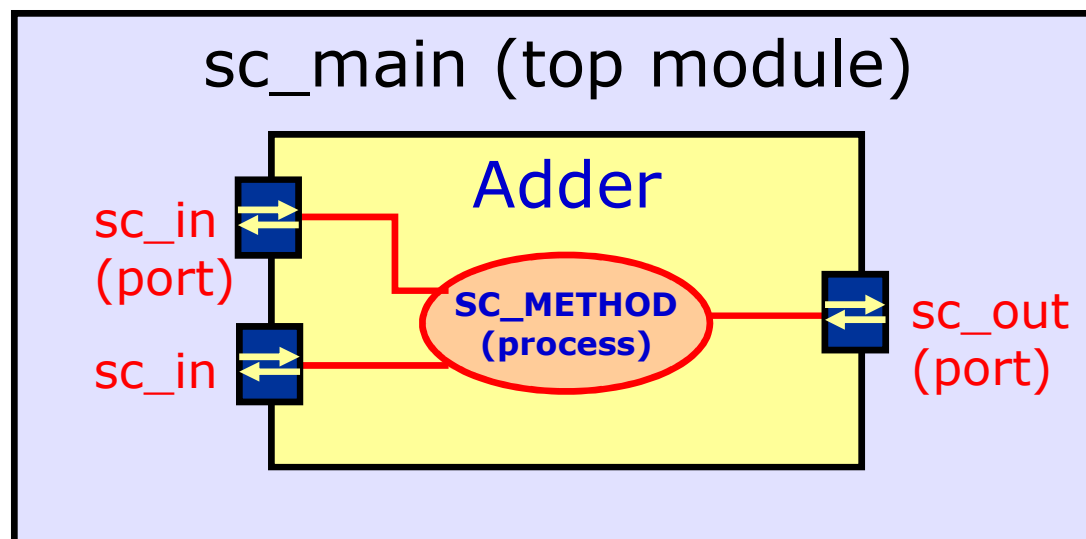


- ▶ `sc_clock` was derived from `sc_module` in 2.0
- ▶ In 2.1, `sc_clock` is derived from `sc_signal<bool>`.

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;
}
```

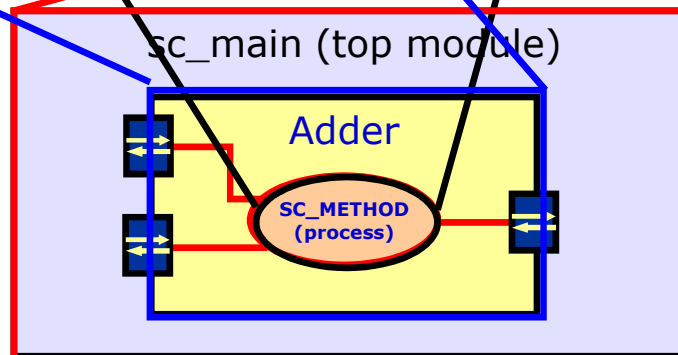
Adder.cpp

```
#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;
}
```

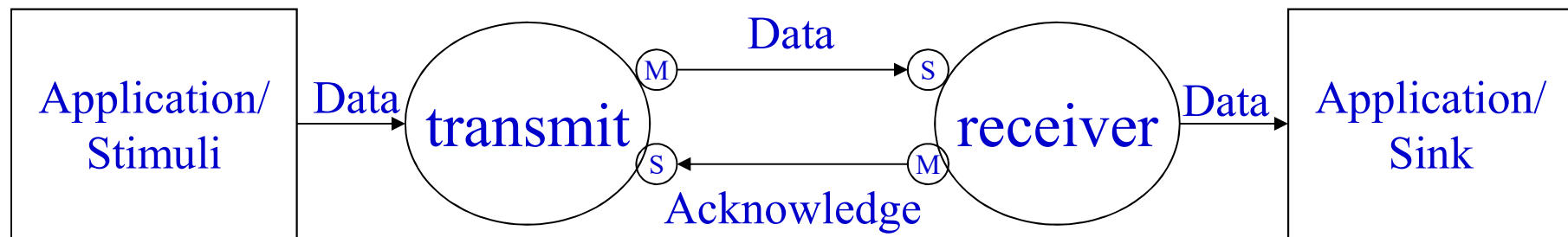
main.cpp





A Not Simple Example

Specification



C/C++ Model



```
frame data;
void transmit(void) {
    int framenum;
    frams s;
    packet buffer;
    event_t event;

    framenum = 1;
    get_data_fromApp(&buffer);
    while (true) {
        s.info = buffer;
        s.seq = framenum;
        send_data_toChannel(&s);
        start_timer(s.seq);
        // If timer times out packet was lost
        wait_for_event(&event);
        if (event == new_frame) {
            get_data_fromChannel(s)
            if (s.ack == framenum) {
                get_data_fromApp(&buffer);
                inc(framenum);
            }
        }
    }
}
```

```
//global data frame storage for Channel
//transmits frames to Channel
// sequence number for framces
// Local frame
// Buffer to hold intermediate data
// Event to trigger actions in transmit

// Initialize sequence numbers
// Get initial data from Application
// Runs forever
// Put data into frame to be sent
// Set sequence number of frame
// Pass frame to Channel to be sent
// Start timer to wait for acknowledge

// Wait for events from channel and timer
// Got a new frame
// Read frame
// Did we get the correct acknowledge
// Yes, get more data
// Increase framenum
```

C/C++ Model - Cont.



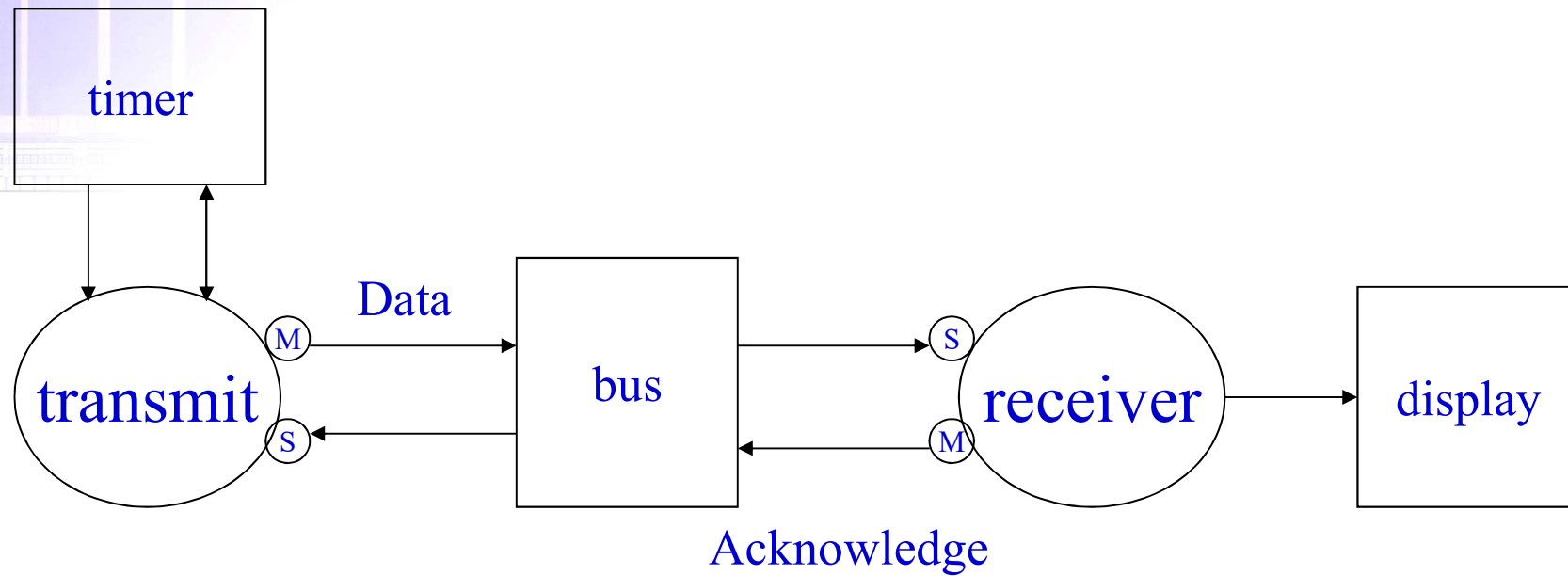
```
void receiver(void) {  
    int framenum;  
    frame r,s;  
    event_t event;  
  
    framenum = 1;  
    while (true) {  
        wait_for_event(&event);  
        if (event == new_frame) {  
            get_data_fromChannel( r );  
            if (r.seq == framenum) {  
                send_data_toApp(&r.info);  
                inc(framenum);  
            }  
            s.ack = framenum - 1;  
            send_data_toChannel(&s);  
        }  
    }  
}
```

// Gets frames from channel
// Scratchpad frame number
// Temp frames to save information
// Event to cause actions in receiver

// Start framenum at 1
// Runs forever
// Wait for data from Channel
// A new frame has arrived
// Get the data from the Channel
// Is this the frame we expected
// Yes, then send data to application
// Get ready for the next frame

// Send back an acknowledge that frame was received
// Send acknowledge

SystemC Model



SystemC Model – packet



```
// packet.h file
#ifndef    PACKETINC
#define    PACKETINC
```

```
#include "systemc.h"
```

```
struct packet_type {
    long info;
    int seq;
    int retry;
```

```
    inline bool operator == (const packet_type& rhs) const
    {
        return (rhs.info == info && rhs.seq == seq &&
                rhs.retry == retry);
    }
```

```
};
```

```
extern
```

```
void sc_trace(sc_trace_file *tf, const packet_type& v,
              const sc_string& name);
```

```
#endif
```

```
// packet.cpp file
#include "packet.h"
```

```
void sc_trace(sc_trace_file *tf, const packet_type& v,
              const sc_string& NAME) {
    sc_trace(tf, v.info, NAME + ".info");
    sc_trace(tf, v.seq, NAME + ".seq");
    sc_trace(tf, v.retry, NAME + ".retry");
}
```

SystemC Model – transmit



```
// transmit.h
#include "packet.h"

SC_MODULE (transmit) {
    sc_in<packet_type>    tpackin;
    sc_in<bool>           timeout;
    sc_out<packet_type>   tpackout;
    sc_inout<bool>        start_timer;
    sc_in<bool>           clock;

    int buffer;
    int framenum;
    packet_type packin, tpackold;
    packet_type s;
    int retry;
    bool start;

    void send_data();
    int get_data_fromApp();

    // Constructor
    SC_CTOR(transmit) {
        SC_METHOD(send_data);
        sensitive << timeout;
        sensitive_pos << clock;
        framenum = 1;
        retry = 0;
        start = false;
        buffer = get_data_fromApp();
    }
};
```

```
// transmit.cpp
#include "transmit.h"
int transmit::get_data_fromApp() {
    int result;
    result = rand();
    cout << "Generate: Sending Data Value = " << result << "\n";
    return result;
}
void transmit::send_data() {
    if (timeout) {
        s.info = buffer;
        s.seq = framenum;
        s.retry = retry;
        retry++;
        tpackout = s;
        start_timer = true;
        cout << "Transmit:Sending packet no. " << s.seq << endl;
    } else {
        packin = tpackin;
        if (!(packin == tpackold)) {
            if (packin.seq == framenum) {
                buffer = get_data_fromApp();
                framenum++;
                retry = 0;
            }
            tpackold = tpackin;
            s.info = buffer;
            s.seq = framenum;
            s.retry = retry;
            retry++;
            tpackout = s;
            start_timer = true;
            cout << "Transmit:Sending packet no. " << s.seq << endl;
        }
    }
}
```

SystemC Model – noisybus



```
// noisybus.h
#include "packet.h"

SC_MODULE (noisybus) {
    sc_in<packet_type> tpackin;
    sc_in<packet_type> rpackin;
    sc_out<packet_type> tpackout;
    sc_out<packet_type> rpackout;

    packet_type packin;
    packet_type packout;
    packet_type ackin;
    packet_type ackout;

    void receive_data();
    void send_ack();

    // Constructor
    SC_CTOR(noisybus) {
        SC_METHOD(receive_data);
        sensitive << tpackin;

        SC_METHOD(send_ack);
        sensitive << rpackin;
    }
};
```

```
// noisybus.cpp
#include "noisybus.h"

void noisybus::receive_data() {
    int i;
    packin = tpackin;
    cout << "Noisybus: Received packet seq no. = " << packin.seq
    << endl;
    i = rand();
    packout = packin;
    cout << "Noisybus: Random number = " << i << endl;
    if ((i > 1000) && (i < 5000)) {
        packout.seq = 0;
    }
    rpackout = packout;
}

void noisybus::send_ack() {
    int i;
    ackin = rpackin;
    cout << "Noisybus: Received Ack for packet = " << ackin.seq <<
    endl;
    i = rand();
    ackout = ackin;
    if ((i > 10) && (i < 500)) {
        ackout.seq = 0;
    }
    tpackout = ackout;
}
```

SystemC Model – receiver



```
// receiver.h
#include "packet.h"

SC_MODULE(receiver) {
    sc_in<packet_type> rpackin;
    sc_out<packet_type> rpackout;
    sc_out<long> dout;
    sc_in<bool> rclk;

    int framenum;
    packet_type packin, packold;
    packet_type s;
    int retry;

    void receive_data();

    //Constructor
    SC_CTOR(receiver) {
        SC_METHOD(receive_data);
        sensitive_pos << rclk;
        framenum = 1;
        retry = 1;
    }
};
```

```
// receiver.cpp
#include "receiver.h"

void receiver::receive_data() {
    packin = rpackin;
    if (packin == packold) return;
    cout << "Receiver: got packet no. = " << packin.seq << endl;
    if (packin.seq == framenum) {
        dout = packin.info;
        framenum++;
        retry++;
        s.retry = retry;
        s.seq = framenum - 1;
        rpackout = s;
    }
    packold = packin;
}
```


SystemC Model – timer



```
// timer.h
#include "systemc.h"

SC_MODULE(timer) {
    sc_inout<bool> start;
    sc_out<bool> timeout;
    sc_in<bool> clock;

    int count;

    void runtimer();

    // Constructor
    SC_CTOR(timer) {
        SC_THREAD(runtimer);
        sensitive_pos << clock;
        sensitive << start;
        count = 0;
    }
};
```

```
// timer.cpp
#include "timer.h"

void timer::runtimer() {
    while(true) {
        if (start) {
            cout << "Timer: timer start detected " << endl;
            count = 5;
            timeout = false;
            start = false;
        } else {
            if (count > 0) {
                count--;
                timeout = false;
            } else {
                timeout = true;
            }
        }
        wait();
    }
}
```

SystemC Model – display



```
// display.h
#include "packet.h"
```

```
SC_MODULE(display) {
    sc_in<long> din;
```

```
    void print_data();
    // Constructor
    SC_CTOR(display) {
        SC_METHOD(print_data);
        sensitive << din;
    }
};
```

```
// display.cpp
#include "display.h"
```

```
void display::print_data() {
    cout << "Display: Data value received, Data = " << din << endl;
}
```

SystemC Model – main



```
// main.cpp
#include "packet.h"
#include "timer.h"
#include "transmit.h"
#include "noisybus.h"
#include "receiver.h"
#include "display.h"

int sc_main(int argc, char* argv[]) {
    sc_signal<packet_type> PACKET1, PACKET2,
    PACKET3, PACKET4;

    sc_signal<long> DOUT;
    sc_signal<bool> TIMEOUT, START;
    sc_clock CLOCK("clock", 20); // transmit clock
    sc_clock RCLK("rclk", 15); // receiver clock

    transmit t1("transmit"); // transmit instantiation
    t1.tpackin(PACKET2); // port to signal binding
    t1.timeout(TIMEOUT);
    t1.tpackout(PACKET1);
    t1.start_timer(START);
    t1.clock(CLOCK); // clocking

    noisybus n1("noisybus"); // noisybus instantiation
    n1.tpackin(PACKET1); // connect to transmit
    n1.rpackin(PACKET3); // port to signal binding
    n1.tpackout(PACKET2); // connect to transmit
    n1.rpackout(PACKET4); // port to signal binding

    receiver r1("receiver"); // receiver instantiation
    r1.rpackin(PACKET4); // connect to noisybus
    r1.rpackout(PACKET3); // connect to noisybus
    r1.dout(DOUT); // port to signal binding
    r1.rclk(RCLK); // clocking

    display d1("display"); // display instantiation
    d1 << DOUT; // signal to port connection

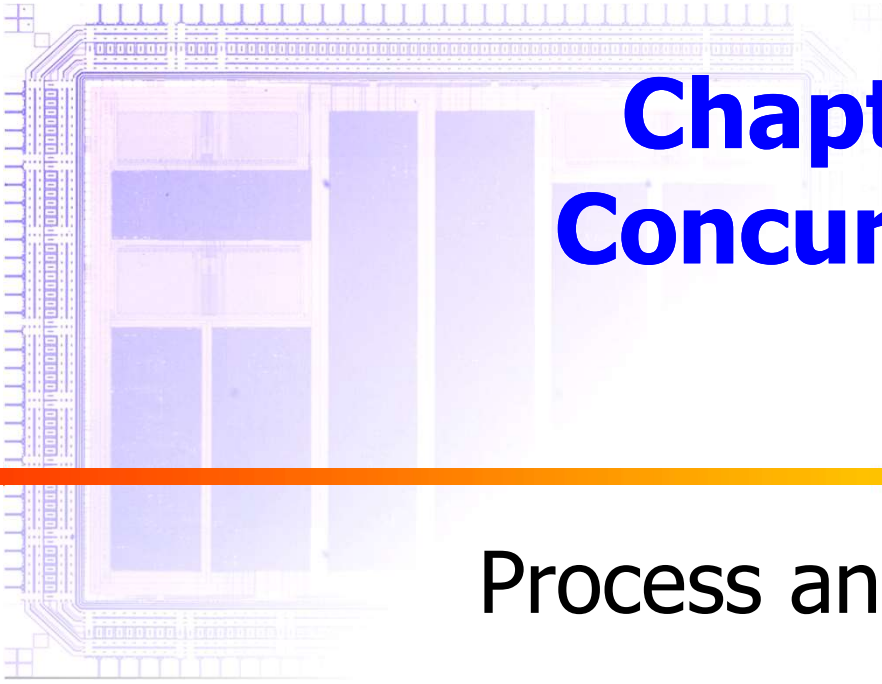
    timer tm1("timer"); // timer instantiation
    tm1 << START << TIMEOUT << CLOCK.signal(); // signal to port connections

    // tracing:
    // trace file creation, with VCD type output
    sc_trace_file *tf = sc_create_vcd_trace_file("simplex");
    // External signals
    sc_trace(tf, CLOCK.signal(), "clock");
    sc_trace(tf, TIMEOUT, "timeout");
    sc_trace(tf, START, "start");
    sc_trace(tf, PACKET1, "packet1");
    sc_trace(tf, PACKET2, "packet2");
    sc_trace(tf, PACKET3, "packet3");
    sc_trace(tf, PACKET4, "packet4");
    sc_trace(tf, DOUT, "dout");

    sc_start(10000); // simulate for 10000 time steps
    // default is ps

    sc_close_vcd_trace_file(tf);

    return(0);
}
```



Chapter 6 Concurrency

Process and Event

Basics

- ▶ In a typical programming language, processes are executed sequentially as control is transferred from one process to another to perform the desired function.
- ▶ Processes are not hierarchical, so no process can call another process directly. Processes can call methods and functions that are not processes.
- ▶ Processes have sensitivity lists. The process is called, or activated, whenever any values in the sensitivity lists are changed.

Basics – Cont.



- ▶ However, hardware components and devices (including wires) are executed in parallel.
- ▶ To mimic the hardware behavior, at each time period, or on the trigger of an event, ready processes are called to simulate the behavior at that moment, till before the next time period, or the next wait().

Event-Driven Simulation in Verilog HDL Simulator



▶ Example code

```
always@ ( in_1 or in_2 ) // model a combinational circuit
    out = in_1 + in_2;    // Adder
```

```
always@ ( posedge clk ) // model a sequential circuit
    q <= d;              // D-FF
```

```
always@ ( event_1 )      // rarely used for HW modeling
```

```
.....
@( event_2 )
```

```
.....
@( event_3 )
```

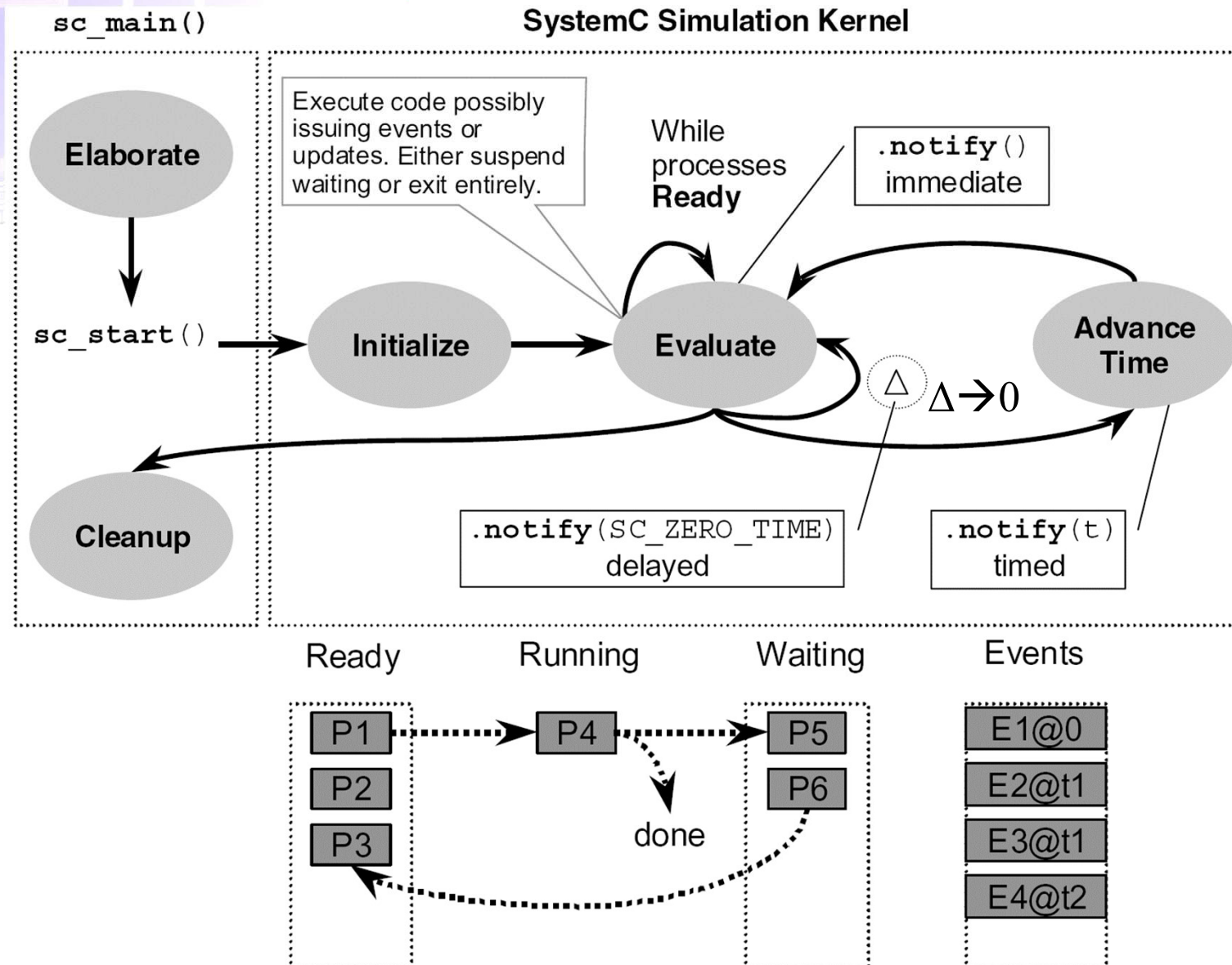
```
.....
```

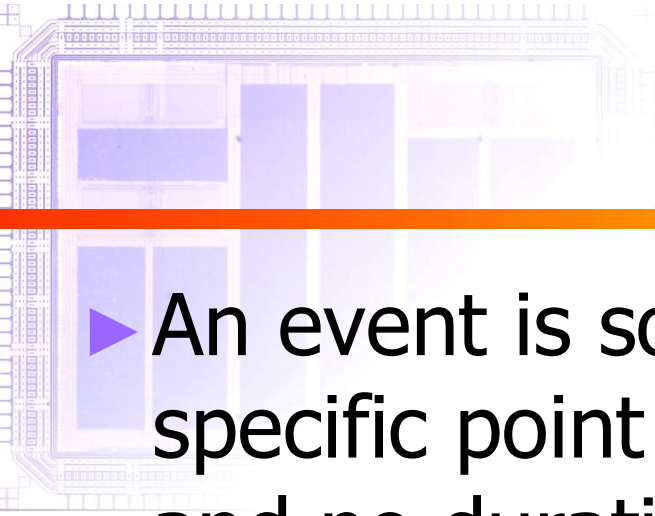
always a simulation instance runs endlessly

@ event-driven keyword

Why does event-driven code (always block) simulate faster than continuous assignment code (assign) ?

Event-Driven Simulation in SystemC Simulation Kernel





sc_event



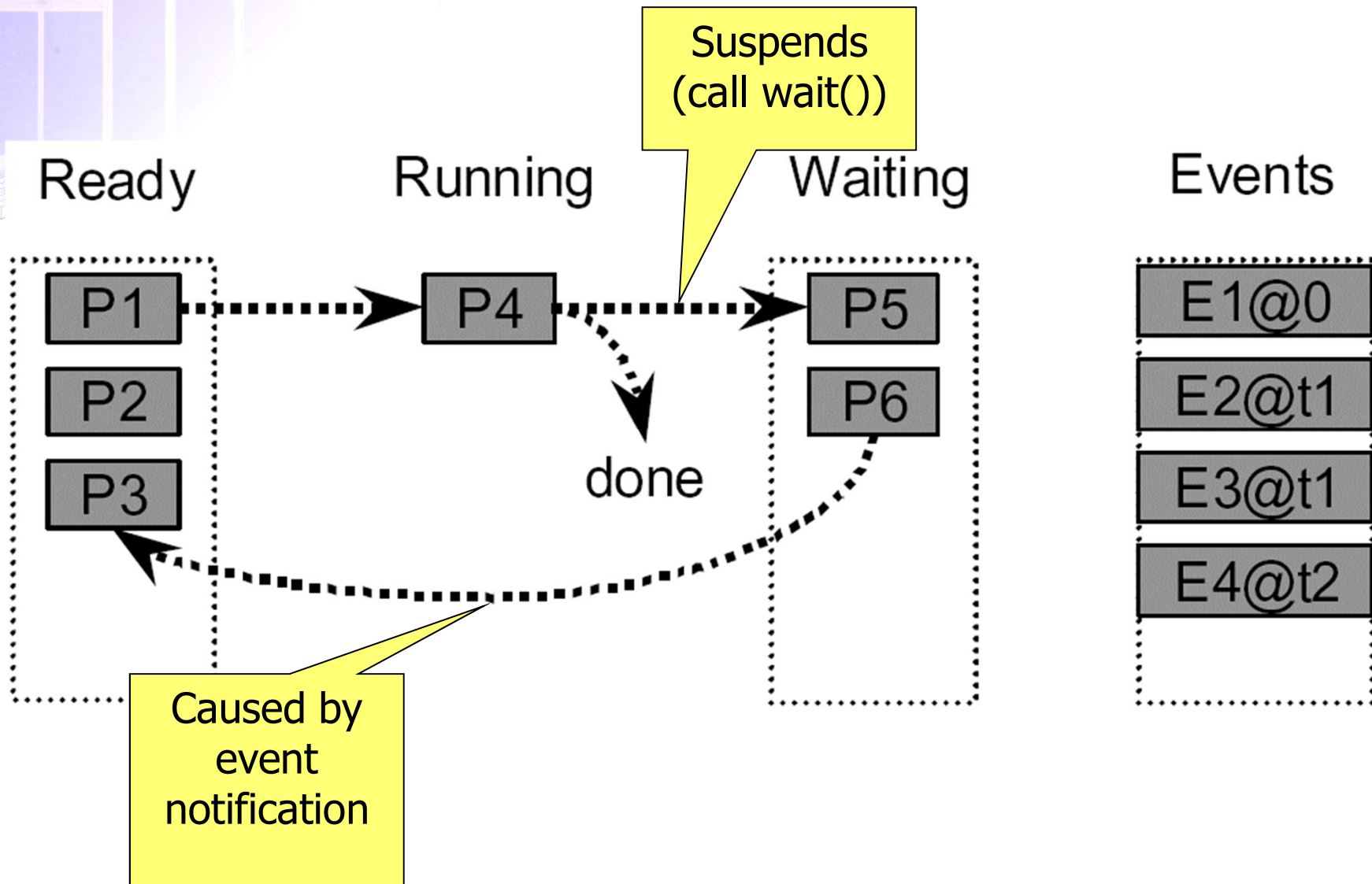
- ▶ An event is something that happens at a specific point in time. An event has no value and no duration
- ▶ You can perform only two actions with an sc_event: wait for it or cause it to occur
- ▶ SystemC lets processes wait for an event by using a dynamic or static sensitivity

Events

- ▶ An event is an object and its synopsis is:

```
class sc_event {  
public:  
    sc_event();  
    ~sc_event();  
    void cancel();  
    void notify();  
    void notify( const sc_time& );  
    void notify( double, sc_time_unit );  
    sc_event_or_list& operator | (const sc_event& ) const;  
    sc_event_and_list& operator & (const sc_event& ) const;  
private:  
    sc_event (const sc_event&);  
    sc_event& operator = ( const sc_event& );  
}
```

Process and Event Pools



Thread

▶ SC_THREAD

▶ Like *initial* in Verilog

▶ Thread is a process that always alive. Unlike method, its local variables are alive throughout the simulation.

▶ When an SC_THREAD process exits, it is gone forever, therefore SC_THREAD **typically contains an infinite loop containing at least one wait**

▶ Thread can be suspended and reactivated. A thread can contain **wait()** statements that suspend the process until an event occurs on one of the signals the process is sensitive to.

SC_THREAD::wait()



Dynamic

```
wait(time);  
wait(event);  
wait(event1 | eventn...); // any of these  
wait(event1 & eventn...); // all of these  
wait(timeout, event); // event with timeout  
wait(timeout, event1 | eventn...); // any event with  
// timeout  
wait(timeout, event1 & eventn...); // all events with  
// timeout  
wait(); // static sensitivity
```

► Ex:

```
...  
sc_event ack_event, bus_error_event;  
...  
wait(t_MAX_DELAY, ack_event | bus_error_event);  
if(timed_out()) break;  
...
```

An Example

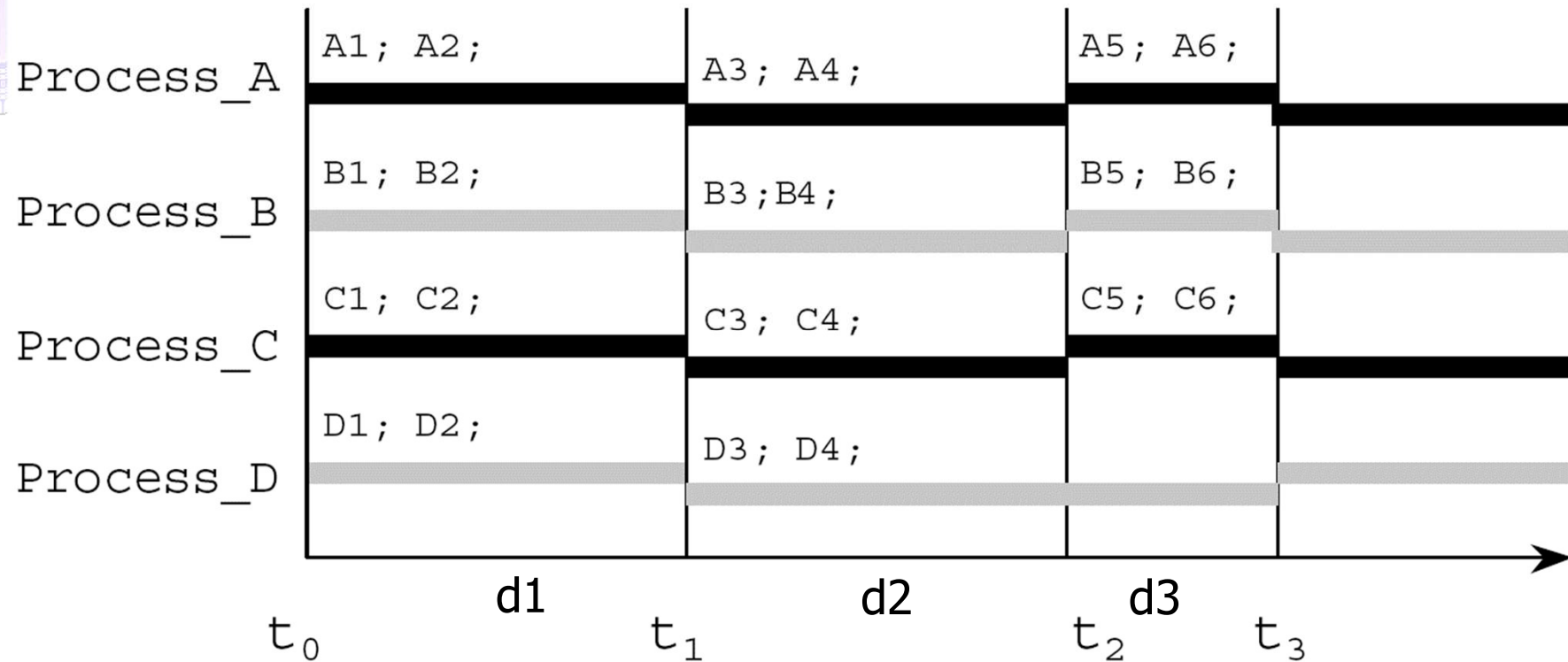
```
Process_A() {  
  //@ t0  
  stmtA1;  
  stmtA2;  
  wait(d1);  
  stmtA3;  
  stmtA4;  
  wait(d2);  
  stmtA5;  
  stmtA6;  
  wait(d3);  
}
```

```
Process_B() {  
  //@ t0  
  stmtB1;  
  stmtB2;  
  wait(d1);  
  stmtB3;  
  stmtB4;  
  wait(d2);  
  stmtB5;  
  stmtB6;  
  wait(d3);  
}
```

```
Process_C() {  
  //@ t0  
  stmtC1;  
  stmtC2;  
  wait(d1);  
  stmtC3;  
  stmtC4;  
  wait(d2);  
  stmtC5;  
  stmtC6;  
  wait(d3);  
}
```

```
Process_D() {  
  //@ t0  
  stmtD1;  
  stmtD2;  
  wait(d1);  
  stmtD3;  
  wait(SC_ZERO_TIME);  
  stmtD4;  
  wait(d2+d3);  
}
```

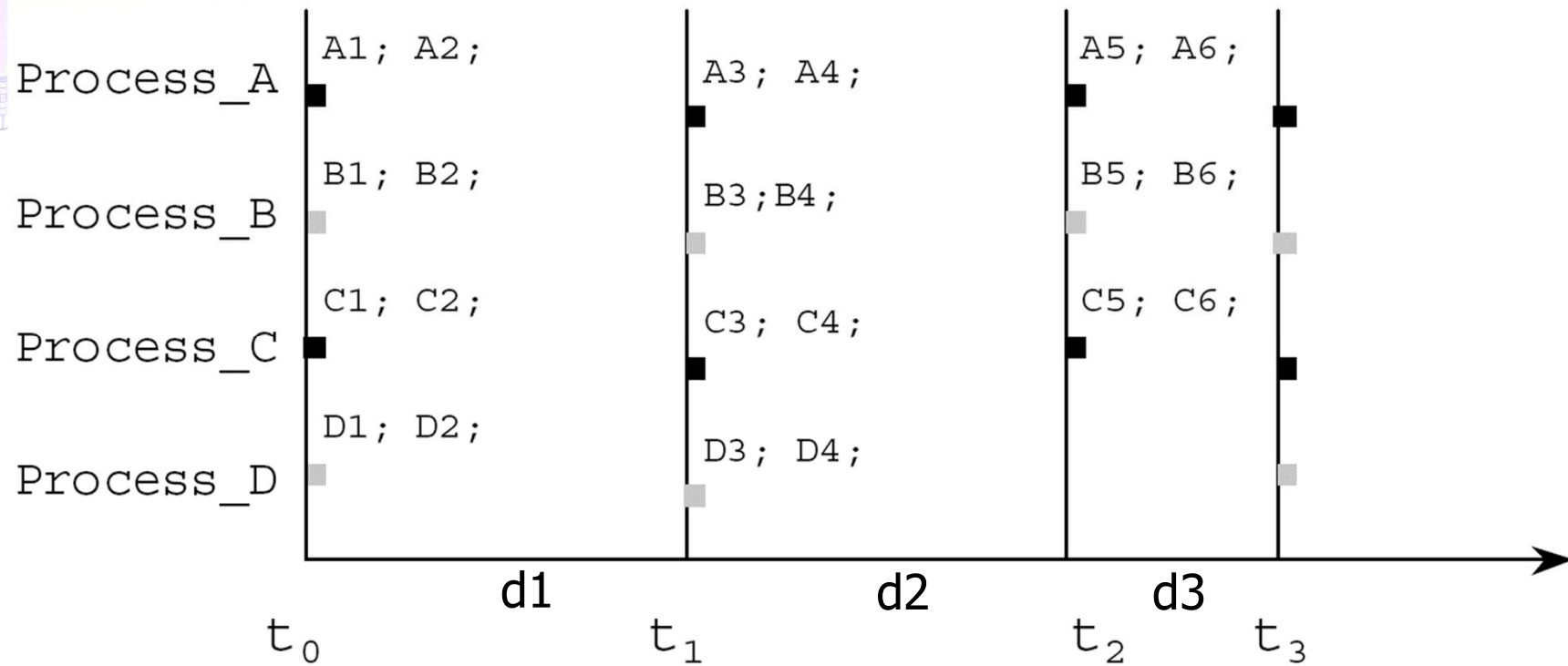
Perceived Simulation Activity



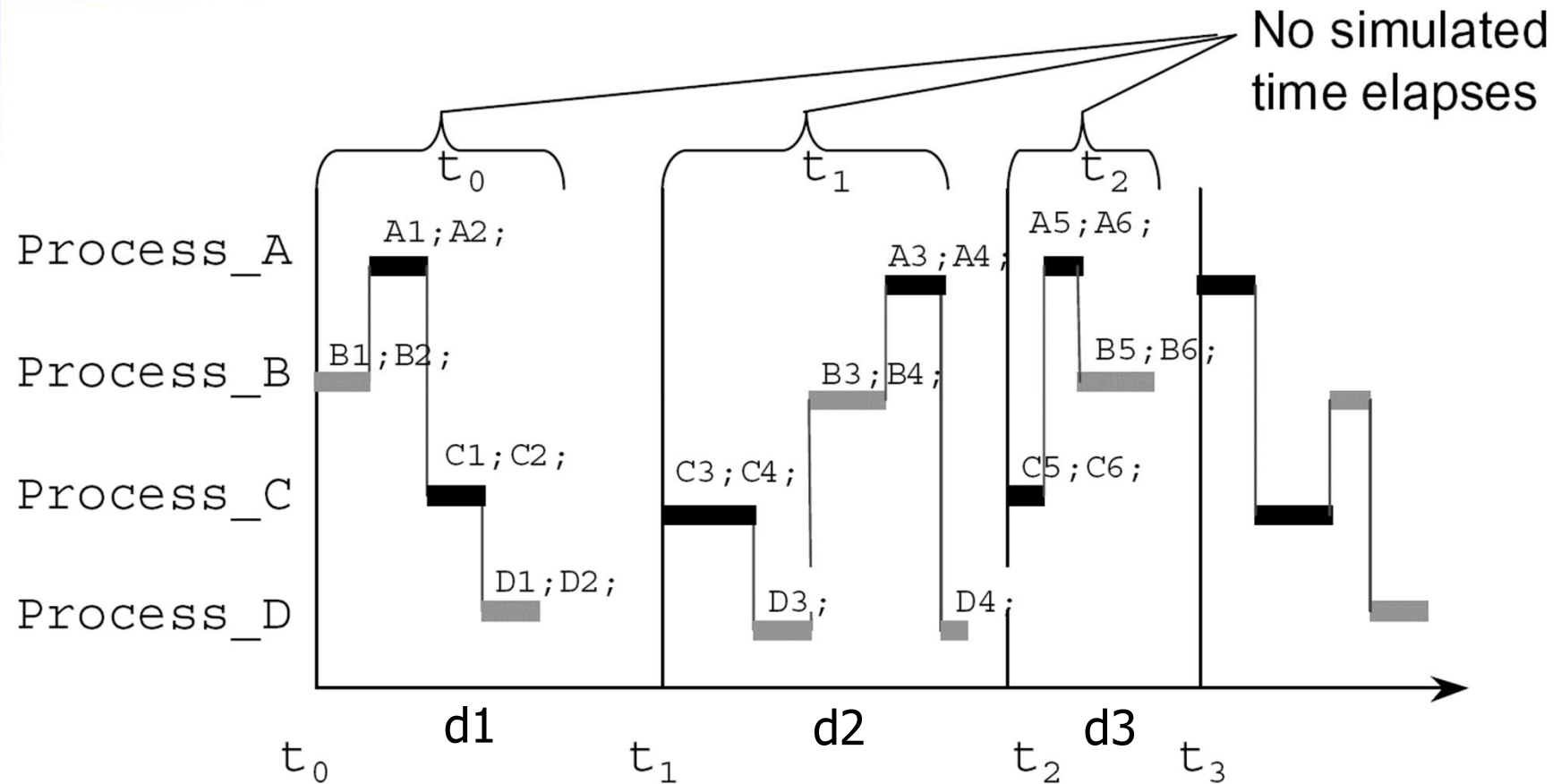
Actual Simulated Activity



清华大学



Simulated Activity with Simulator Time Expanded



Template of SC_THREAD with Static Sensitivity



```
SC_MODULE(synchronous_module) {  
    sc_in<bool> clock;  
  
    void thread();  
};
```

```
    SC_CTOR(synchronous_module) {  
        SC_THREAD(thread);  
        sensitive << clock.pos();  
    }  
    ...  
};
```

// registration the constructor
// registration thread as a process
// sensitivity list

```
void synchronous_module::thread() {  
    for (;;) {  
        wait(); // Resume on positive edge of clock  
        ...  
    }  
}
```

// Member function called once only

Triggering Events: .notify()

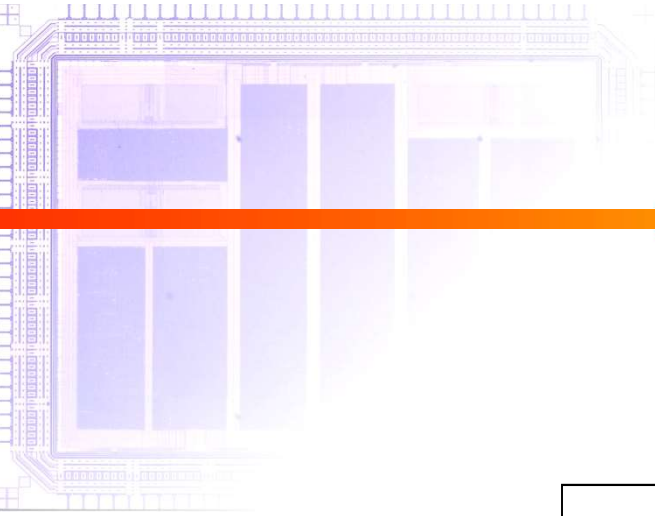


```
// Object-oriented style (preferred)
event_name.notify(); //immediate notification
event_name.notify(SC_ZERO_TIME); // delayed
                                // notification
event_name.notify(time); //timed notification

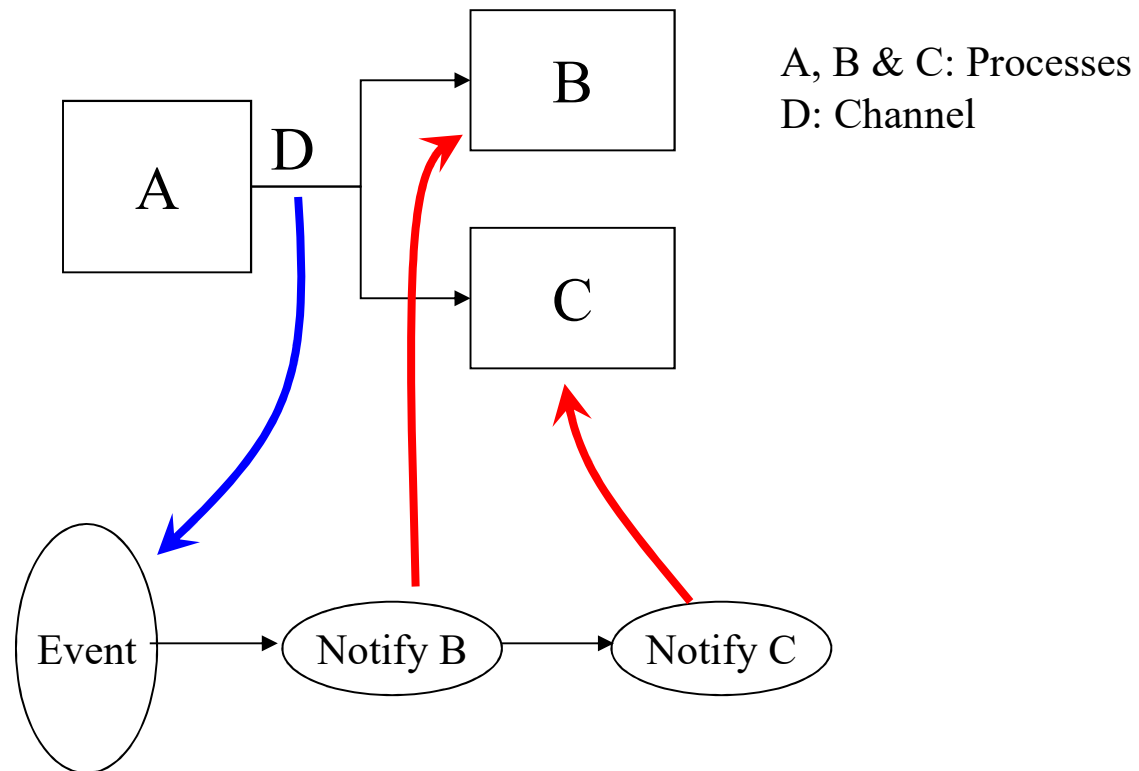
// Functional-call style
notify(event_name); //immediate notification
notify(event_name, SC_ZERO_TIME) ; // delayed
                                // notification
notify(event_name, time); //timed notification
```

► Cancel event

- `event_name.cancel();`



Event



Notify an Event



► 3 ways to notify an event:

- Immediate: the event is triggered in the current evaluation phase of the current delta-cycle; `notify()`
- Delta-cycle delayed: the event will be triggered during the evaluate phase of the next delta-cycle; `notify(0, SC_NS)` or `notify(SC_ZERO_TIME)`
- Timed: the event will be triggered at the specified time in the future; `notify(10, SC_NS)`

```
sc_event my_event;  
sc_time t(10, SC_NS); // a 10ns time interval  
my_event.notify();    // immediate  
my_event.notify(0);   // delta-cycle delayed  
my_event.notify(t);   // notification in 10ns
```

Multiple Event Notification



- ▶ Earlier notification will always override one scheduled to occur later, and an immediate notification is always earlier than any delta-cycle delayed or timed notification.
- ▶ Notice a **potential non-deterministic** situation:

Process A { my_event.notify(); }	Process B { my_event.notify(0); }	Process C { wait(my_event); }
--	---	-------------------------------------

- ▶ If A first then B, C will be executed at both current and next delta-cycle.
- ▶ However if B first then A, C will execute once only at the current delta-cycle

Canceling Event Notifications



- ▶ A pending delayed event notification may be canceled using `cancel()`. However immediate event cannot be canceled.

```
sc_event a, b, c;  
sc_time t(10, SC_MS);  
a.notify();  
notify(0, b);  
notify(t, c);
```

```
a.cancel();      // Error!  
b.cancel();      // Canceled!  
c.cancel();      // Canceled!
```



Multiple Pending Events

- ▶ While `sc_event()` can only allow a single pending event, `sc_event_queue()` can handle multiple pending events.

▶ **SC_METHOD**

- ▶ Method is just SystemC ways of saying function, or subroutine. Therefore it inherits the behavior of a function.
- ▶ A process is called to execute and returns the execution back to the calling mechanism when completed.
- ▶ A locally declared variable is not permanent. Meaning, the value of a local variable is no longer valid after the end of a method.

More about SC_METHOD



- ▶ SC_METHOD process is similar to Verilog `always@` block or VHDL `process`
- ▶ SC_METHOD process cannot issue a `wait()` or any other blocking methods. Only SC_THREAD / SC_CTHREAD process can.
- ▶ Be carefull when you use `read()/write()` methods of `sc_fifo` data type (A kind of channel). These are blocking methods!

Dynamic Sensitivity for SC_METHOD: next_trigger();



```
next_trigger(time);  
next_trigger(event);  
next_trigger(event1 | eventi...); //any of these  
next_trigger(event1 & eventi...); //all of these  
                                //required  
  
next_trigger(timeout, event);    //event with timeout  
next_trigger(timeout, event1 | eventi...); //any + timeout  
next_trigger(timeout, event1 & eventi...); //all + timeout  
next_trigger(); //re-establish static sensitivity
```

Template of SC_METHOD



```
SC_MODULE(module)
{
    sc_in<bool> input_signal;

    void method();

    SC_CTOR(module) {
        SC_METHOD(method);
        sensitive << input_signal;
    }
    ...
};

void module::method() { // called whenever sensitive signal is changed
    ...                // actions
}
```

Clocked Thread



▶ **SC_CTHREAD**

- ▶ Clocked thread is a special case of a thread process. The difference lies in that a clocked thread is only sensitive to one edge of a clock cycle (positive or negative).
- ▶ We suggest not to use clock threads. It can be fully replaced by a normal thread.
- ▶ **wait_until()** is a function ~~only~~ can be used in clocked threads. It halts the execution of the process until a specific event has occurred.

Template of SC_CTHREAD



清華大學

```
SC_MODULE( synchronous_module ) {
    sc_in<bool> clock;
    sc_in<bool> reset;

    void CT();

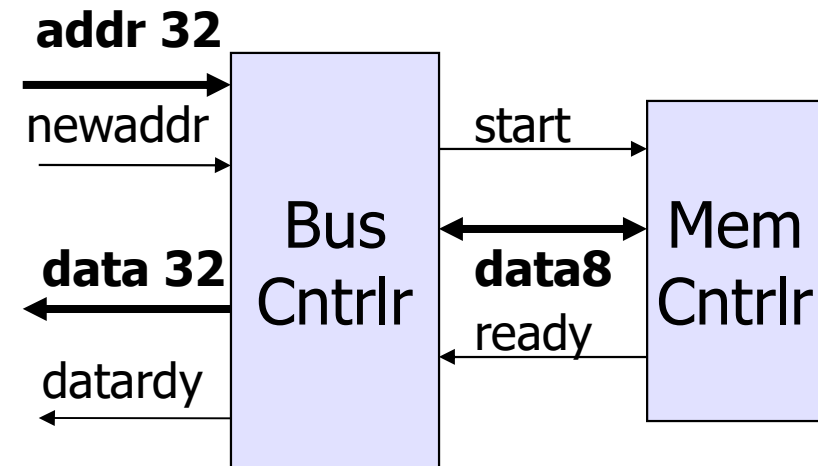
    SC_CTOR( synchronous_module ) {
        SC_CTHREAD( CT, clock.pos() );
        reset_signal_is(reset, true); // add this if reset is required
    }
    ...
};

void synchronous_module::CT() {
    if (reset) {
        ... // reset actions
    }
    while(true) {
        wait(1); // Wait for 1 clock cycle
        ... // clocked actions
    }
}
```

Clocked Thread Example: Bus Controller



```
SC_MODULE(bus) {  
    sc_in_clk clock;  
    sc_in<bool> newaddr;  
    sc_in<sc_uint<32> > addr;  
    sc_in<bool> ready;  
    sc_out<sc_uint<32> > data;  
    sc_out<bool> start;  
    sc_out<bool> datardy;  
    sc_inout<sc_uint<8> >  
        data8;  
    sc_uint<32> tdata;  
    sc_uint<32> taddr;  
  
    void xfer();  
}
```



```
SC_CTOR(bus) {  
    SC_CTHREAD(xfer,  
        clock.pos());  
    datardy = true;  
}  
};
```

Clocked Thread Example: Bus Controller (cont'd)



```
void bus::xfer() {  
    while (true) {  
        wait_until(  
            newaddr.delayed() ==  
            true);  
        taddr = addr.read();  
        datardy = false;  
        data8 = taddr.range(7,0);  
        start = true;  
        wait();  
        data8 = taddr.range(15,8);  
        start = false;  
        wait();  
        data8 = taddr.range(23,16);  
        wait();
```

```
        data8 = taddr.range(31,24);  
        wait();  
        wait_until(ready.delayed()  
            == true);  
        tdata.range(7,0)=data8;  
        wait();  
        tdata.range(15,8)=data8;  
        wait();  
        tdata.range(23,16)=data8;  
        wait();  
        tdata.range(31,24)=data8;  
        data = tdata;  
        datardy = true;  
    }  
}
```


More about SC_THREAD and SC_CTHREAD (1)



- ▶ A function associated with such process instance is called once and only once by the kernel, except when a clocked thread process is reset.
- ▶ Only thread or clocked thread process can call the function `wait()`. Such a call causes the calling process to suspend execution. Method process will result in runtime error.
- ▶ The process instance is resumed when the kernel causes the process to continue execution starting with the statement immediately following the most recent call to function `wait()`.
- ▶ When a thread or clocked thread process is resumed, the process executes until it reaches the next call to function `wait()`. Then, the process is suspended once again.

More about SC_THREAD and SC_CTHREAD (2)

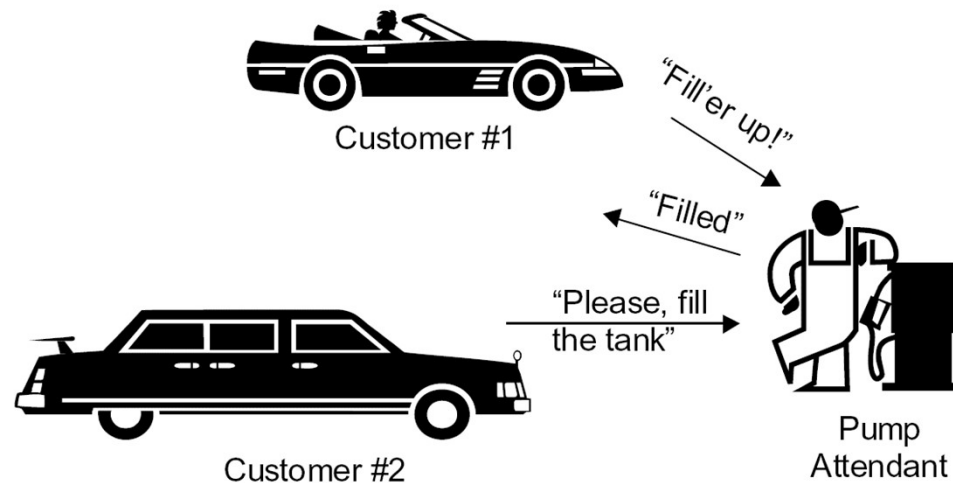


- ▶ A thread process instance may have static sensitivity, or may call function wait to create dynamic sensitivity. A clocked thread process instance is statically sensitive only to a single clock.
- ▶ Each thread process requires its own execution stack. As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.
- ▶ If the thread or clocked thread process executes the entire function body or executes a return statement and thus returns control to the kernel, the associated function shall not be called again for that process instance. The process instance is then said to be terminated.

Example – Gas Station



Early Gas Station



```
SC_MODULE(gas_station) {  
  
    ...  
    sc_event e_request1, e_request2;  
    sc_event e_filled;  
  
    // Constructor  
    SC_CTOR(gas_station){  
        SC_THREAD(customer1_thread);  
        sensitive(e_filled);  
  
        SC_THREAD(customer2_thread);  
  
        SC_METHOD(attendant_method);  
        sensitive << e_request1 << e_request2;  
        dont_initialize();  
    }//endconstructor gas_station  
  
    // Declare processes  
    void customer1_thread(void);  
    void customer2_thread(void);  
    void attendant_method(void);  
};
```

Implementation



```
...
void gas_station::customer1_thread(void) {
    for (;;) {
        // Simulate gas tank emptying time
        wait(EMPTY_TIME);
        cout << " Customer1 needs gas" <<
            endl;
        m_tank1 = 0;
        do {
            e_request1.notify(); // I need fillup!
            wait(); // use static sensitivity
            // Somebody got filled
        } while (m_tank1 == 0);
        // We got filled
    } //endforever
} //end customer1_thread()
...
```

```
void gas_station::attendant_method(void) {
    if (!m_filling) {
        ...
        cout << " Filling tank" << endl;
        next_trigger(FILL_TIME);
        m_filling = true;
        ...
    } else {
        ...
    } //endif

    e_filled.notify(SC_ZERO_TIME); // We
    finished filling!
    m_filling = false; // go back to waiting
} //endif
} //end attendant_method()
```

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.

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.
- ▶ 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.

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

References

- ▶ SystemC Version 2.0 User's Guide, Update for SystemC 2.0.1
- ▶ SystemC 2.0.1 Language Reference Manual
- ▶ Draft Standard SystemC Language Reference Manual –SystemC 2.1 LRM, 4/25/2005
- ▶ TLM Library
- ▶ L. M. Ayough, A. H. Abutalebi, O. F. Nadjarbashi, S. Hessabi, "Verilog2SC: A Methodology for Converting Verilog[®] HDL to SystemC"
- ▶ <http://www.systemc.org>
- ▶ David C. Black and Jack Donovan, *SystemC: From the Ground Up*, Springer, 2004.
- ▶ L. Cai and D. Gajski, "Transaction Level Modeling: an overview," *CODES+ISSS'03*.
- ▶ 李昆忠、簡韶逸、鄺獻榮、邱瀝毅、郭致宏，電子系統層級設計教授，教育部顧問室「超大型積體電路與系統設計教育改進」計畫SLD聯盟。