SystemC Overview and Datatype

Kun-Chih (Jimmy) Chen 陳坤志

kcchen@nycu.edu.tw

Institute of Electronics,
National Yang Ming Chiao Tung University

SystemC Overview

What is SystemC

- SystemC is a modeling platform
  - A set C++ class library to add hardware modeling constructs
  - Simulation kernel
  - Supports different levels of abstraction
    - Untimed Functional Model
    - Transaction Level Model (TLM)
    - Bus Function Model

- SystemC is
  - A library of C++ classes
    - Processes (for concurrency)
    - Clocks (for time)
    - Hardware data types
    - Wait and watching (for reactivity)
    - Modules, ports, signals (for hierarchy)
    - Abstract ports and protocols (abstract communication)

SystemC vs. 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, standard template library (STL) and such.
- However, you need to follow SystemC methodology closely to make sure the simulation executes correctly.

SystemC vs. 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 vise versa. Reason: Verilog and VHDL do not support transaction-level.

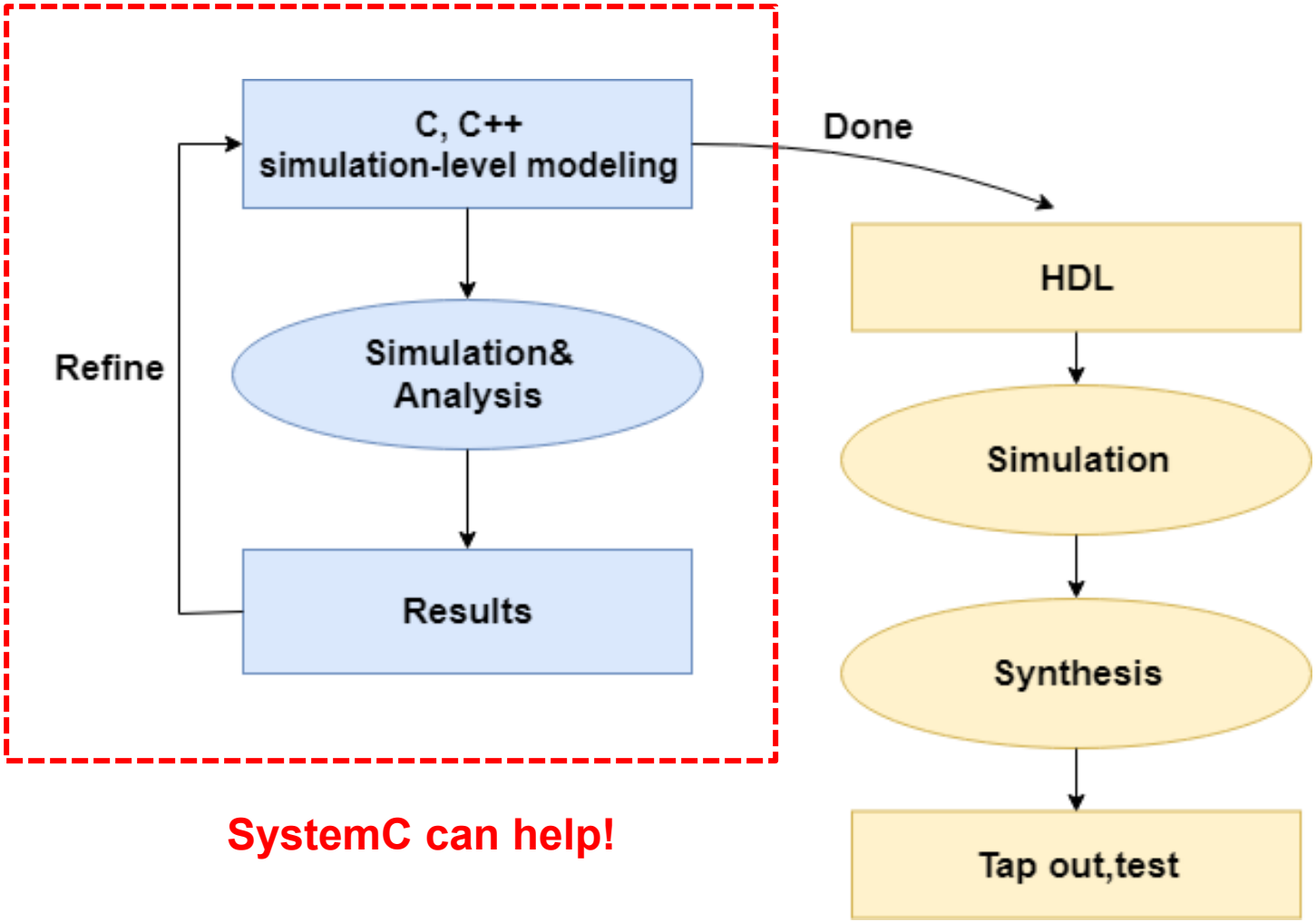
Why do we need SystemC? (1/2)

- The increasingly shortened time to market requirements
  - Verify the design in early time
- The growing complexity
  - Integration of devise devices

Therefore…

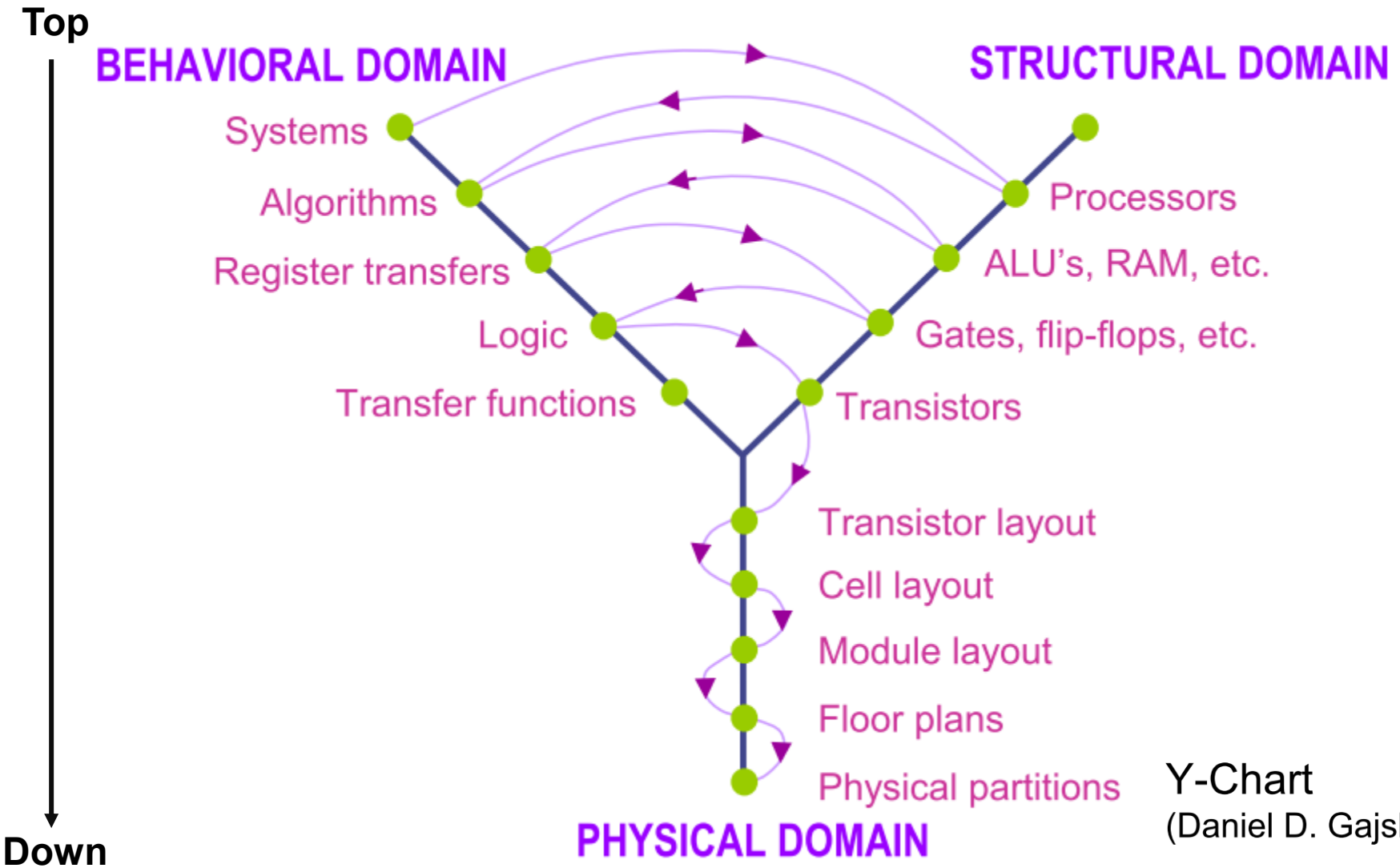
- High abstraction level simulation for hardware.
- Build a platform to co-verify hardware & software. (i.e., Synopsys PA)
- Decreasing the risk of project failure.

Why do we need SystemC? (2/2)



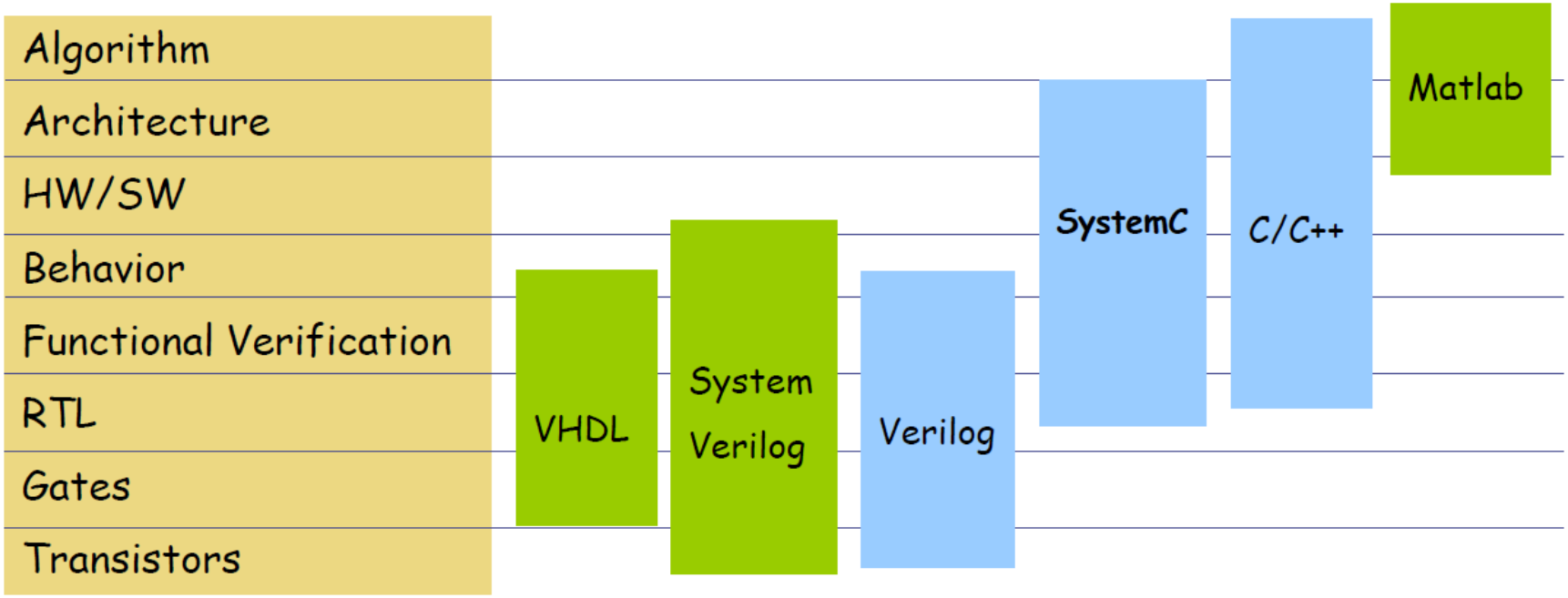
SystemC can help!

Design Domains and Levels



Y-Chart
(Daniel D. Gajski, 1988)

What levels does SystemC play for?



SystemC becomes a popular HDL since 2005

電子工程專輯 > EDA/IP

EDA/IP RSS

SystemC逐漸成為IC設計的通用語言

上網時間: 2005年06月27日 打印版 SHARE 字型大小: A A

關鍵字：SLD SystemC IC設計 system-level design 系統級設計

tool!--BROKEN_TABLE_HIDDEN_BEGIN

BROKEN_TABLE_ HIDDEN_END -->

參加設計自動化研討會(DAC)的設計師們共聚一堂，就比暫存器傳輸級(RTL)抽象等級更高的IC設計軟體工具之前景和缺陷等問題交換了看法。電子系統級設計(SLD)包含許多組成元素，利用類似SystemC的軟體代碼來定義用於開發積體電路的結構模型。SystemC已被談論了許多年，專家們表示如今該語言正逐漸成為IC設計通用語言。

「SystemC使多級協同設計及測試在許多等級上成為可能——應用、作業系統和元件驅動器及RTL。」意法半導體(ST) SoC平台自動化技術部Charles Pilkington表示。盡管SystemC已經入侵成功，Pilkington仍希望能改進SystemC，包括能在語言中增強模擬及除錯定義。

「SystemC很好，雖然並不完美。」IBM公司Thomas J. Watson研究中心系統級設計部研究員Reinaldo Bergamaschi如此說。他對該語言允許用同一種語言寫模型/分析和演算法表示歡迎。他並呼籲開放研發團體應該更投入這樣的運作，並以更多類似於Linux的捐獻方式來開放SystemC平台。他表示，SystemC工具和方法論之間也需要在目前RTL流程下有更好的聯繫。

方便管理65nm SoC複雜性是另一個轉向SystemC的原因。飛利浦半導體公司技術與標準總監Ralph von Vignau表示：「我們正向更靈活的65nm結構轉移，需要更多多工和自動化，這些結構需要針對功率、性能和範圍進行最佳化。」此外，Spirit聯盟主席Von Vignau則指出，可以幫助加速產品上市也是業界轉向SystemC的一個原因。

目前，開放式SystemC處理級建模(TLM，transaction-level modeling)正獲得像飛利浦半導體、意法半導體和IBM這種大型公司開發人員的支援。TLM標準1.0版定義了應用編程介面(API)和庫，設計用於促進IP共享與多工，加速EDA工具開發，並使得OEM能夠更容易地使用TLM。

SystemC becomes a popular HDL since 2005

電子工程專輯 > EDA/IP

EDA/IP RSS

SystemC促使SoC設計走向更高抽象層

上網時間: 2005年11月02日 打印版 SHARE 字型大小: A A

關鍵字：SystemC SoC 抽象層 開放原始碼 RTL

當SystemC語言以一種新的開放原始碼語言在1999年問世時，給設計工程師中帶來了不小的困惑。什麼是SystemC？一種硬體設計語言？如果是的話，怎麼能是以C++為基礎的呢？一種行為級語言？那麼它為什麼又這麼像RTL？它會不會取代Verilog和VHDL？其關鍵問題就在於它到底是做什麼的？

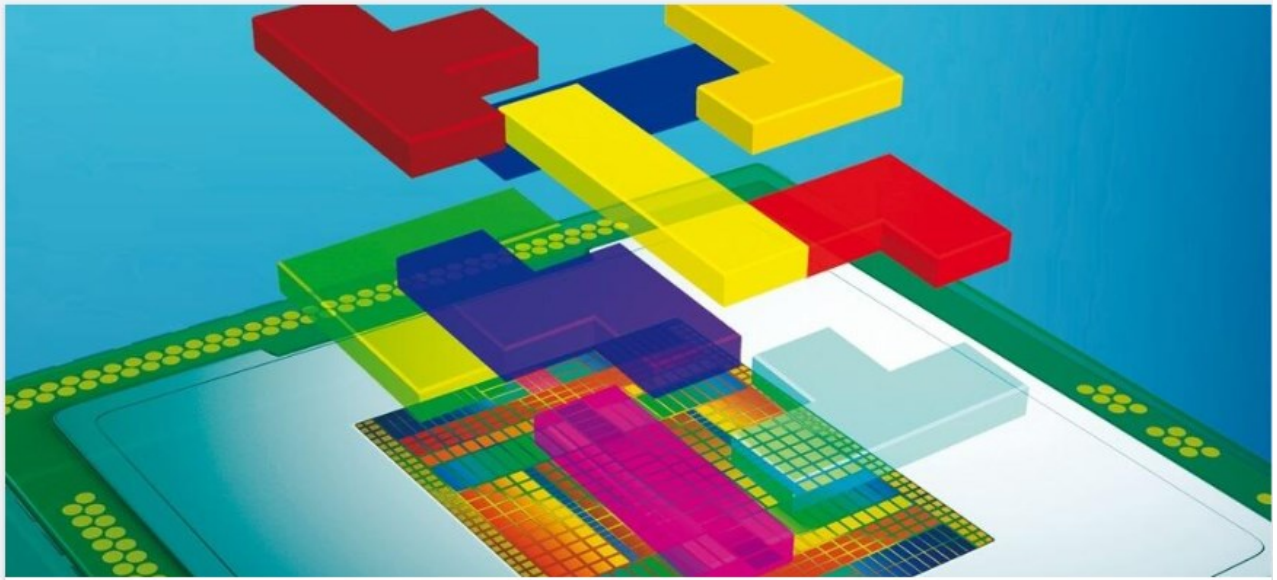
五年過去了，以上的所有問題都有答案了。不僅SystemC存活下來了，而且它在SoC設計流程中的優點也被認可了。從最初的只被歐洲和日本的少數架構設計者所採用到現在廣泛應用於北美的設計者，許多成功應用SystemC進行設計公司和組織已經將它融入到他們的設計流程中。

OSCI(Open SystemC Initiative)組織網站的點選率和人們對SystemC語言的興趣迅速上升，目前SystemC的專利數量已超過22,000而且還在增加中。很明顯SystemC已經成功定位，再不被認為是HDL的替代者，而是連接建構系統行為級模型的系統架構師和編寫RTL建置程式碼的工程師之間的橋樑。正是因為它從新的2.1版本開始加強了事務級的建模能力，SystemC才能跨越這兩個世界而且幫助進行軟硬體協同模擬。更可喜的是，OSCI和OCPIP(Open Core Protocol International Partnership)正為建置一種能共享的建模結構而繼續合作，如果成功將會使第三方的IP更容易整合在SystemC環境中。OCPIP是一個產業的聯盟，致力於製作一種通用的IP介面。現在已經有不少SystemC的電子系統級(ESL)工具和方法問世，使人們夢寐已久的系統級設計流程得以實現。

SystemC becomes a popular HDL since 2005

AI時代的EDA工具進化挑戰

2019年10月1日 • Judith Cheng，EE Times Taiwan



對於AI的「特定領域」(domain specific)架構特性，Mentor的Sawicki則認為高階合成技術(HLS)是最佳化設計方法；他指出，HLS能支援對AI應用十分關鍵的架構探索，特別是與記憶體相關的配置以及功耗分析。Mentor的Catapult HLS平台則能提供設計工程師以標準化的ANSI C++與SystemC語言來描述功能意圖，以高速度產生高品質的RTL；這將大幅降低AI設計的驗證成本。Catapult HLS平台還包括系統整合所需的FPGA展示器(demonstrator)、CPU子系統、軟硬體介面與HLS加速器範本；Nvidia的Tegra X1晶片就是Catapult HLS平台的成功設計案例之一。

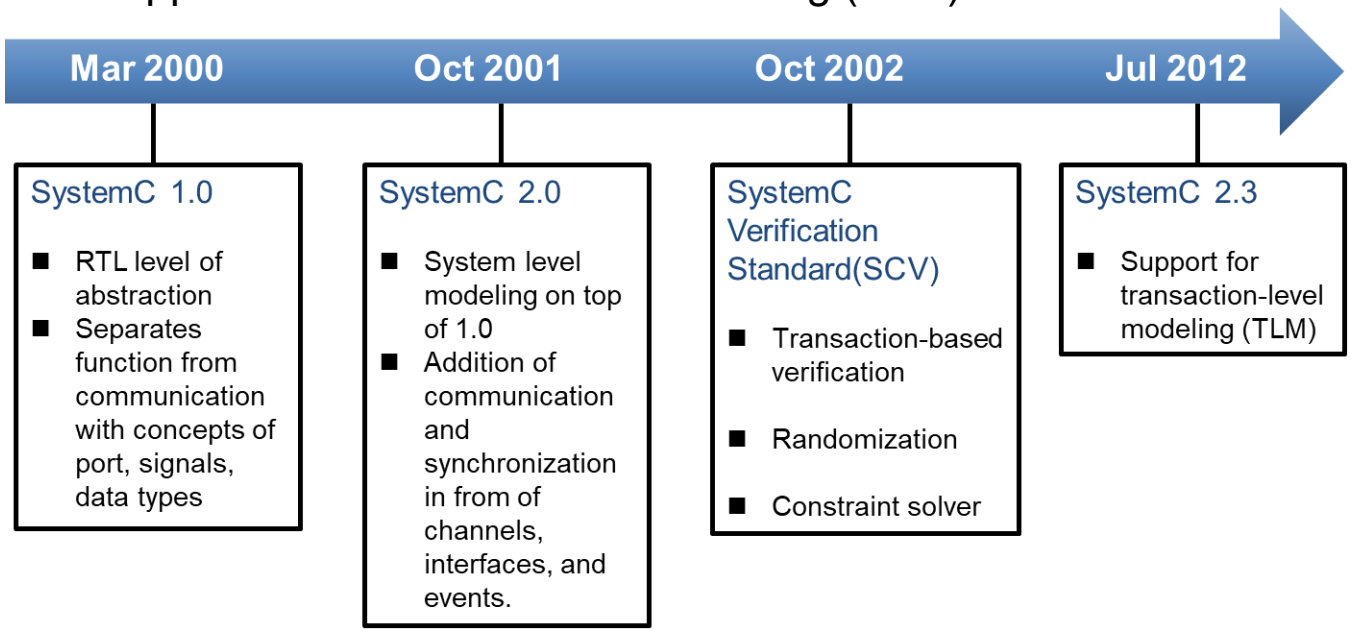
SystemC Myth

- 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 a unified design language be realized?
  - We just don't know. But people are talking about UML, the Unified Modeling Language.

Evolution of SystemC

- Version 1
  - It is just another HDL, not much to do with system-level design.
- Version 2
  - With the adding of channel, now it is a serious system-level language
    - Signal, Mutex, FIFO
  - Add some programming features
- Version 2.3
  - Support for transaction-level modeling (TLM)



SystemC 2.0 Language Architecture

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, Static Dataflow, etc.

Primitive Channels Signal, Mutex, Semaphore, FIFO, etc.	
Core Language Modules Ports Processes Interfaces Channels Events	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 Standard

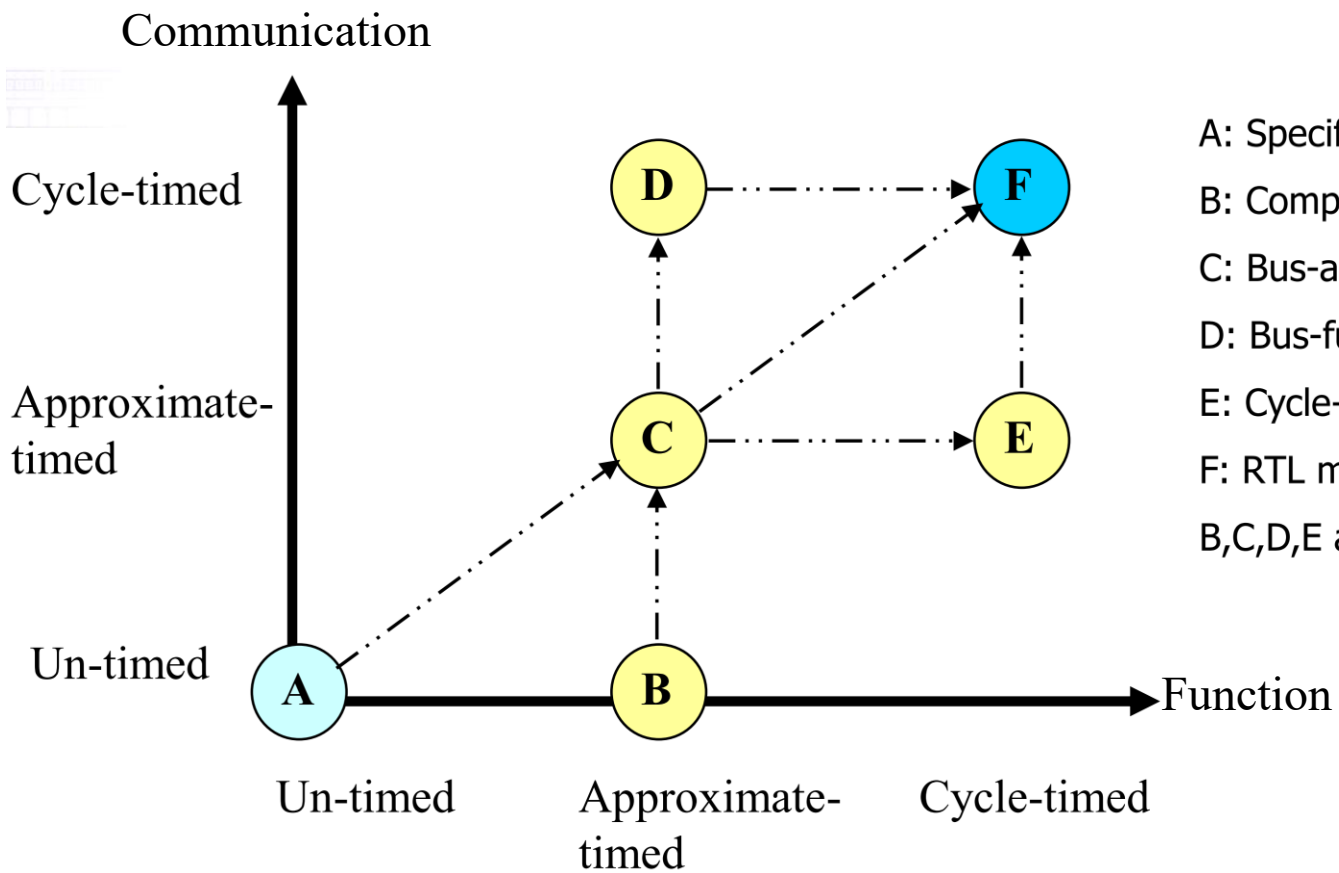
SystemC 2.3 Language Architecture

Methodology-Specific Libraries Master/Slave Library, etc.	**Layered Libraries** Verification Library, TLM Library, etc.

Transaction-Level Modeling	
Primitive Channels Signal, Mutex, Semaphore, FIFO, etc.	
Core Language Modules Ports Processes Interfaces Channels Events	**Data Types** 4-valued Logic Type 4-valued Logic Vectors Bits and Bit Vectors Arbitrary Precision Integers Fixed-Point Types C++ User-Defined Types
Event-Driven Simulation Events, Processes	

C++ Language Standard

Different Abstract Models (1/2)



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

Different Abstract Models (2/2)

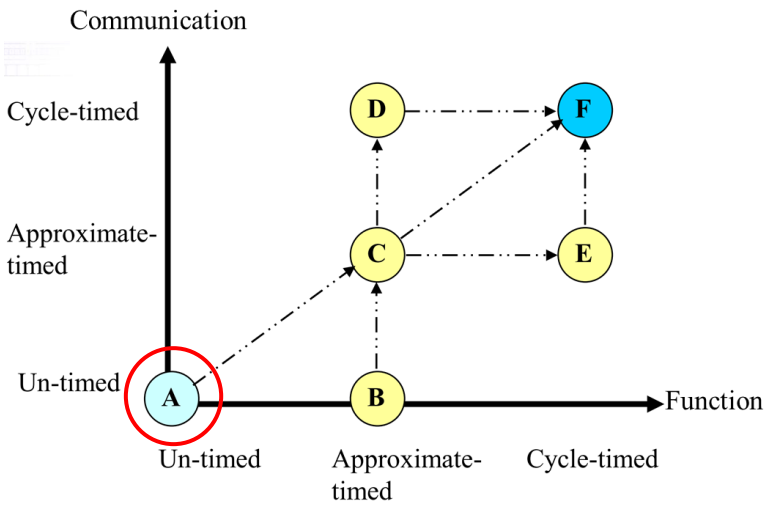
- Three degrees of time accuracy
  - Un-timed computation/communication
    - pure functionality of the design without any implementation details
  - Approximate-timed computation/communication
    - contains system-level implementation details, such as the selected system architecture, the mapping relations 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

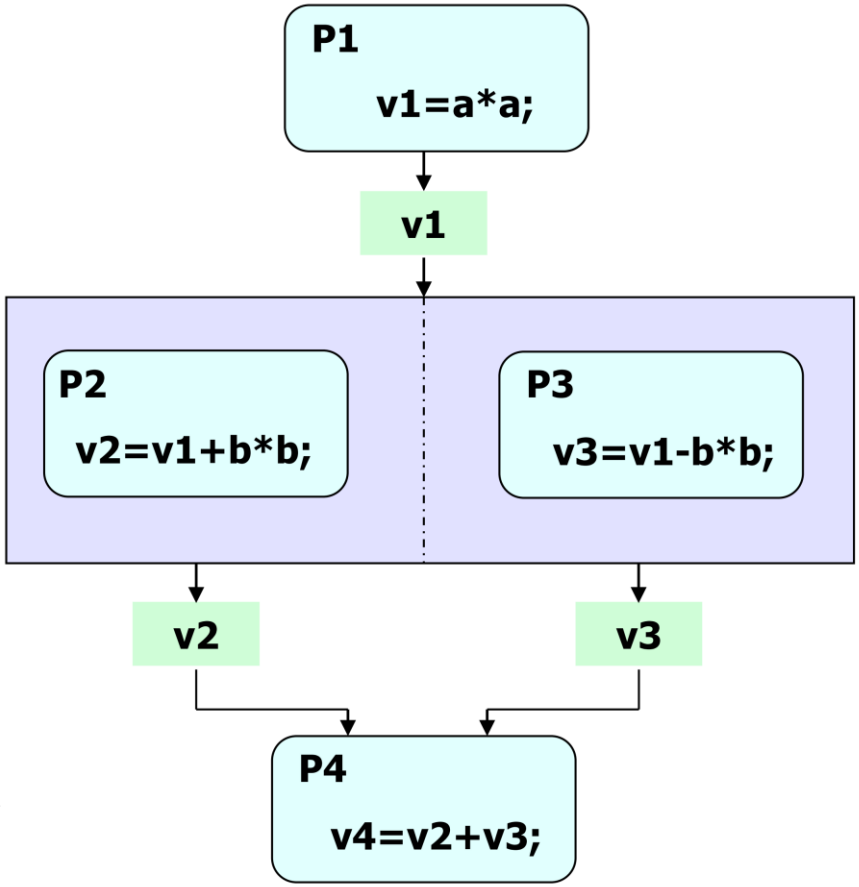
Specification Model (A)

- The functionality of the system is specified in this abstraction level.

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

v1, v2 and v3: variables



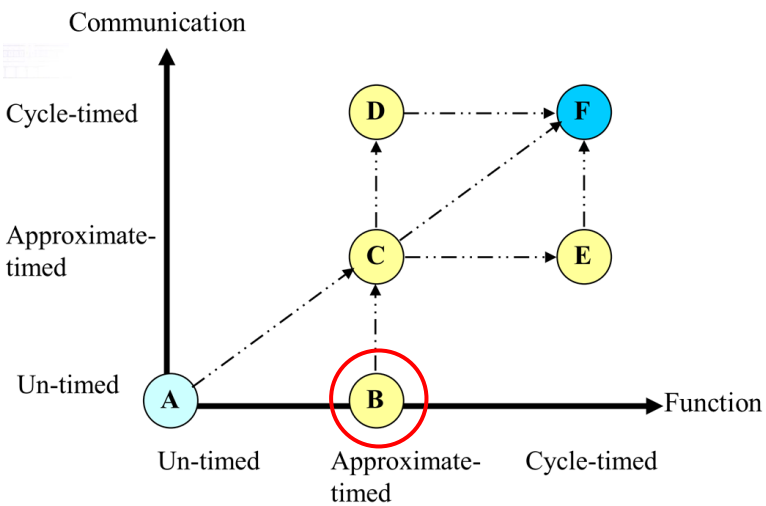


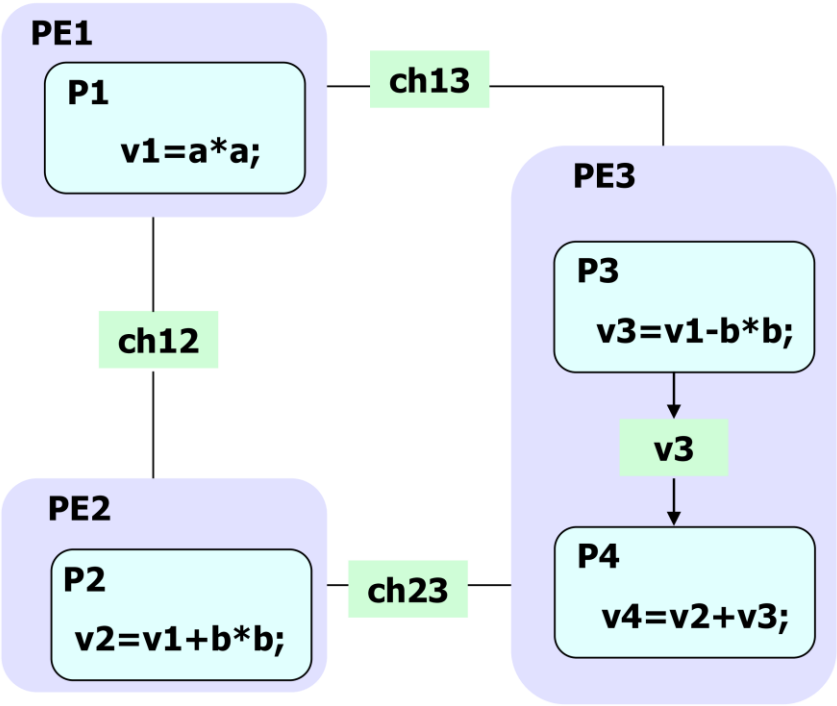
Component-Assembly Model (B)

- The system may be composed of CPU,DSP or other IPs. The system architecture can estimate the computational time without consideration of the communication time. The number of required processing elements is determined in this level.

Ch12, ch13 and ch23: channels

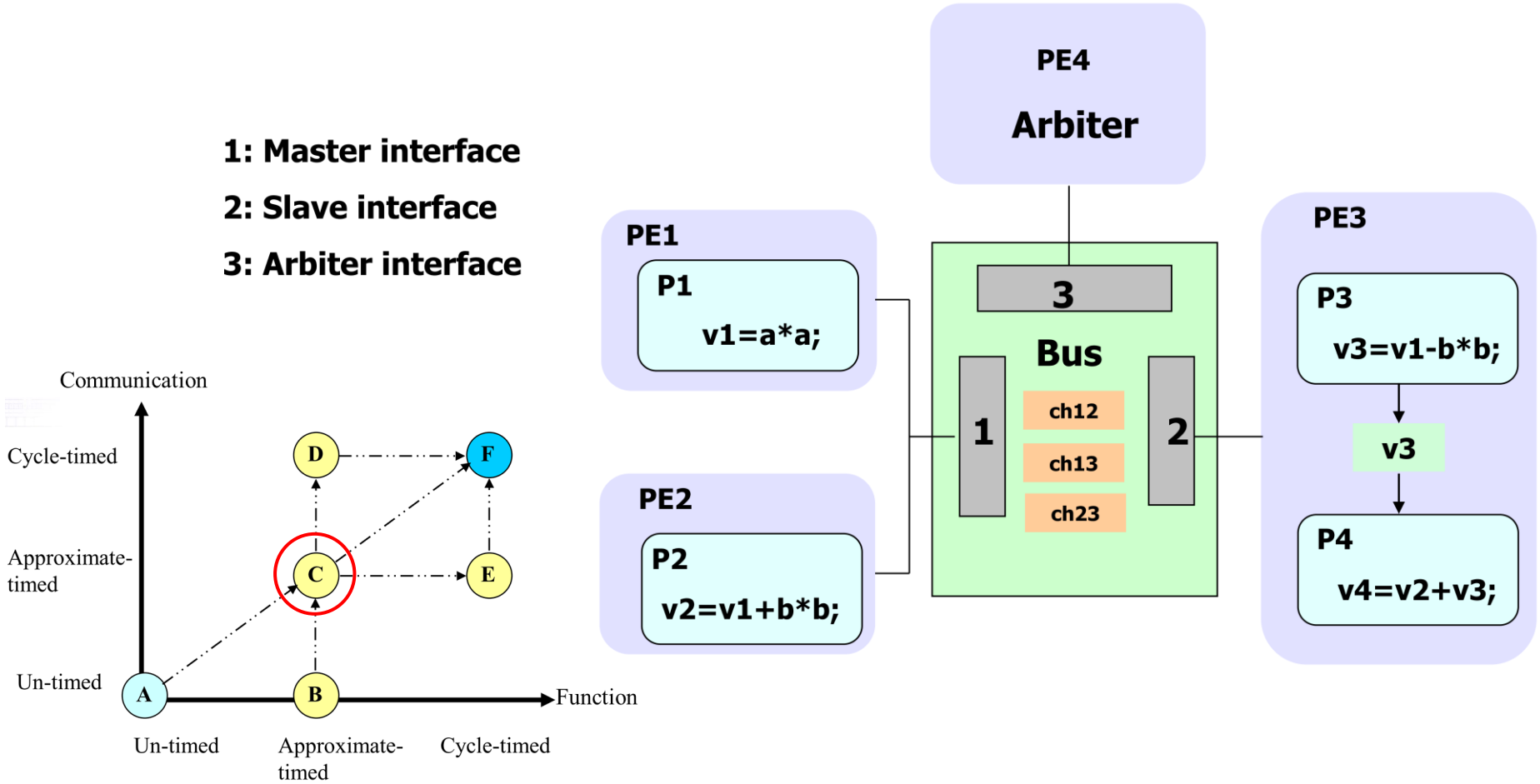
PE1, PE2 and P3: Processing element





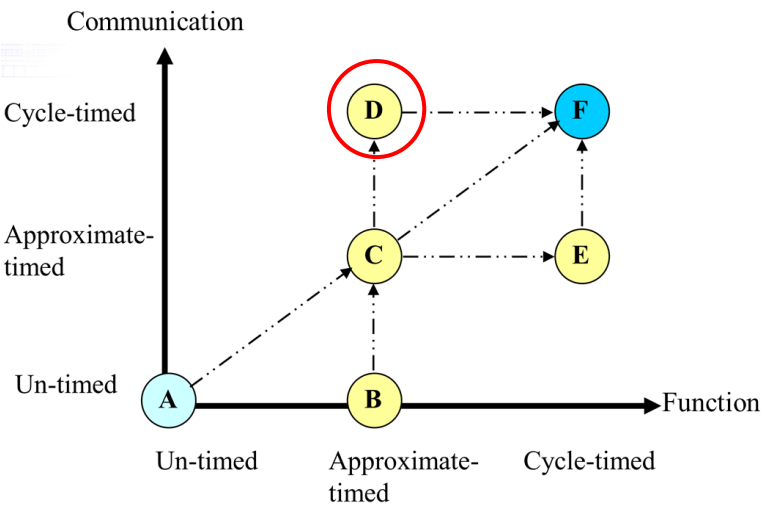
Bus Arbitration Model (C)

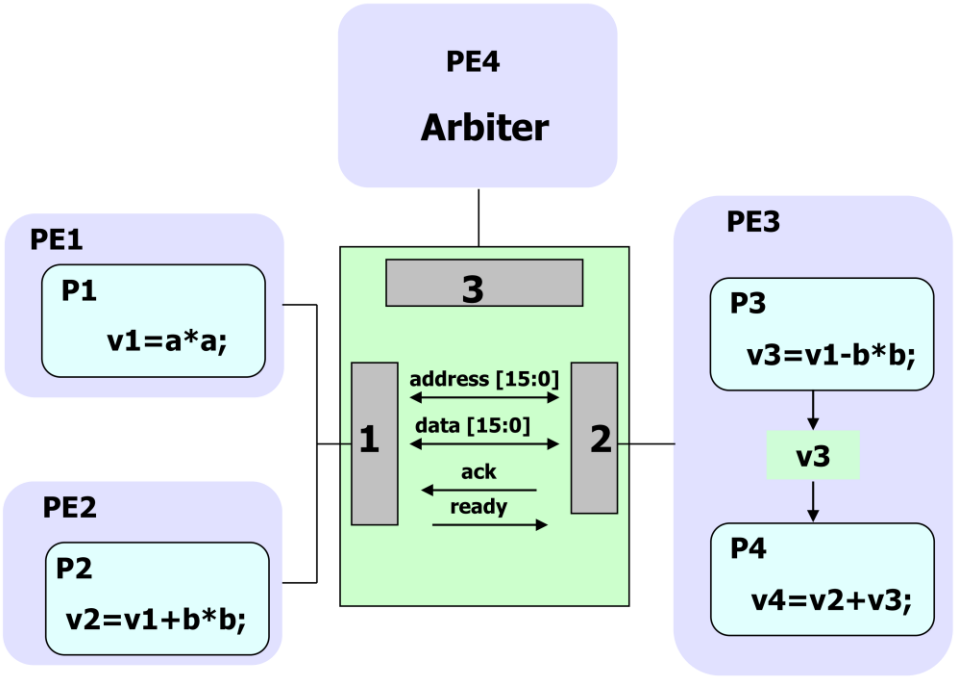
- The data transfer is implemented by the message-passing channel without cycle-accuracy, pin-accuracy and specific detailed protocol.



Bus Functional Model (D)

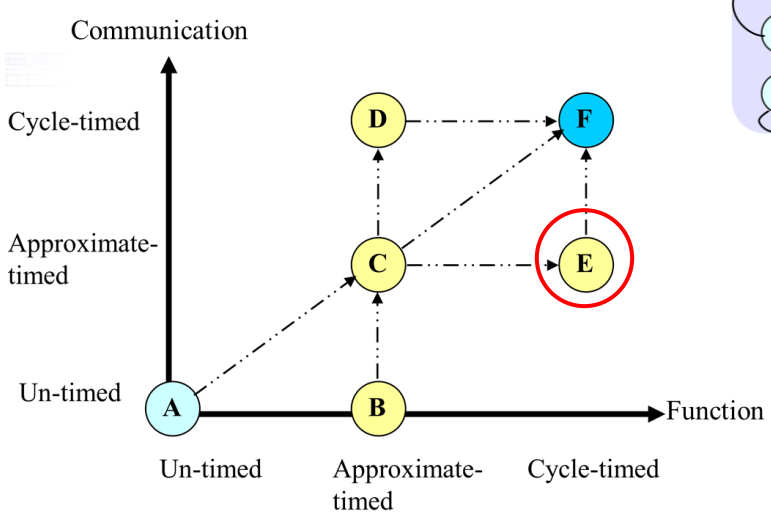
- The message-passing channels are replaced by protocol channels via the procedure of protocol refinement.
- The protocol channels are both cycle-accurate with the specific protocol.

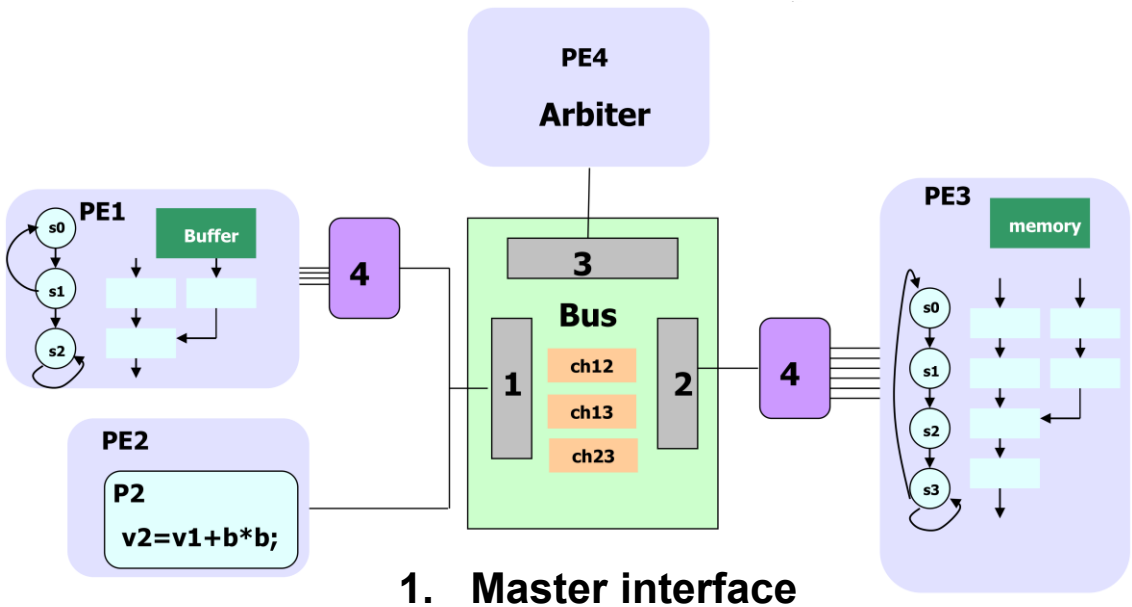




Cycle-Accurate Computation Model (E)

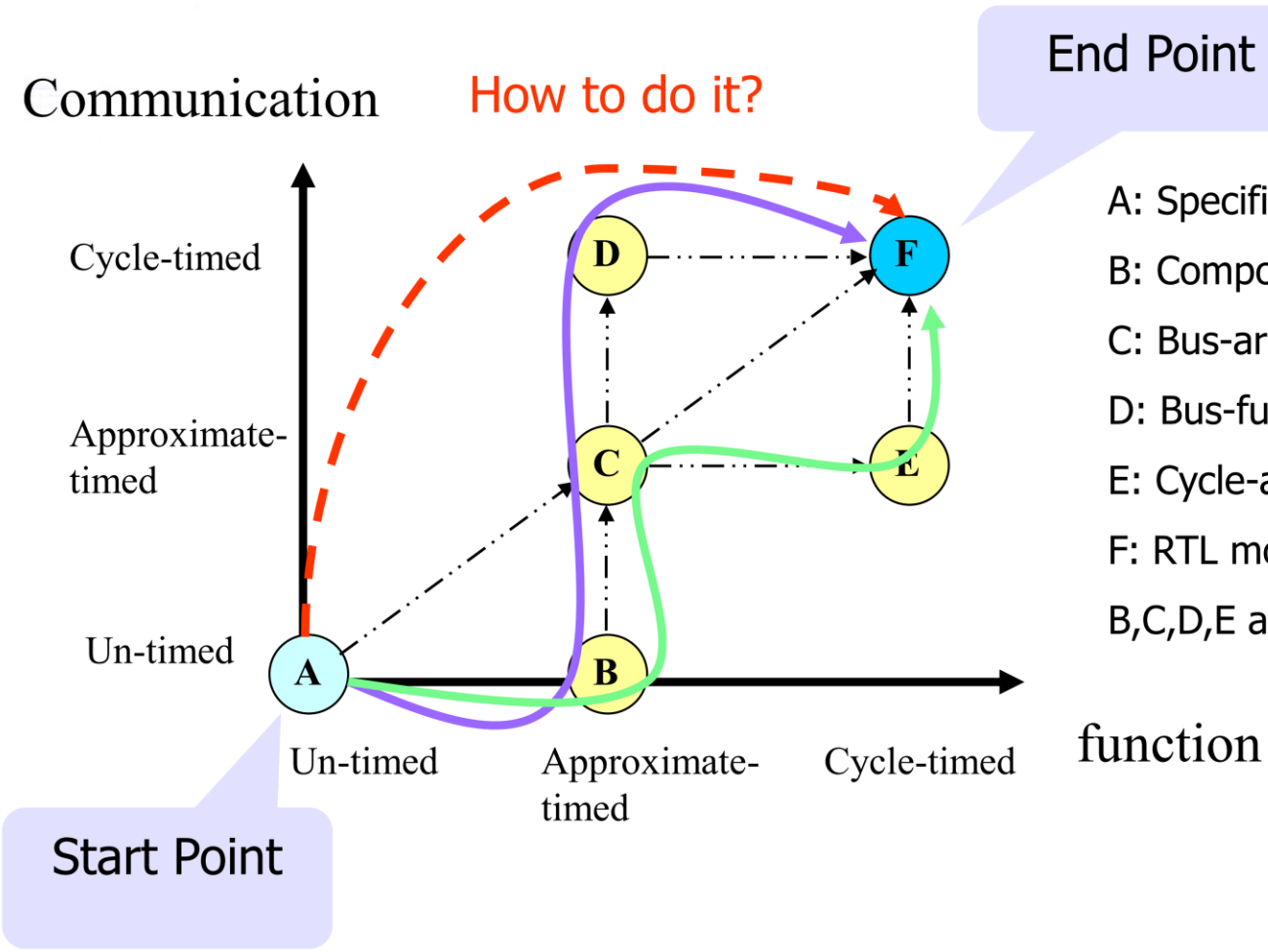
- The processing elements (PE) or IPs are cycle-accurate which may be RTL models.
- The converters or adapters are required to convert data transfer between the higher abstraction channel model and lower abstraction PE or IP models.





1. **Master interface**
2. **Slave interface**
3. **Arbiter interface**
4. **Wrapper/Adapter**

TLM Design Flow



A: Specification model

B: Component-assembly model

C: Bus-arbitration model

D: Bus-functional model

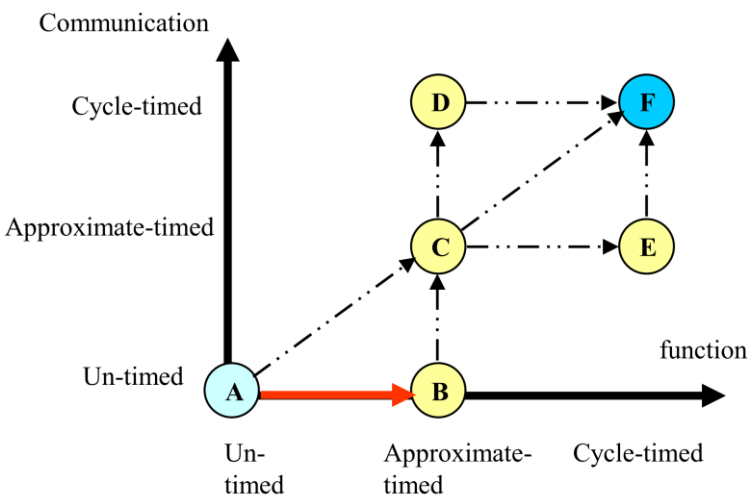
E: Cycle-accuracy computation model

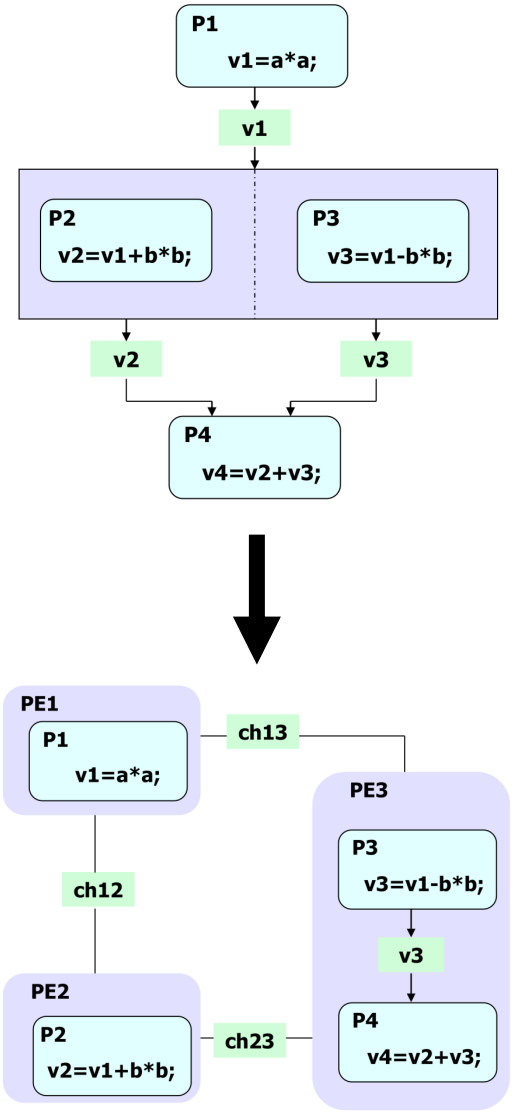
F: RTL model

B,C,D,E are TLMs

Component Assembly

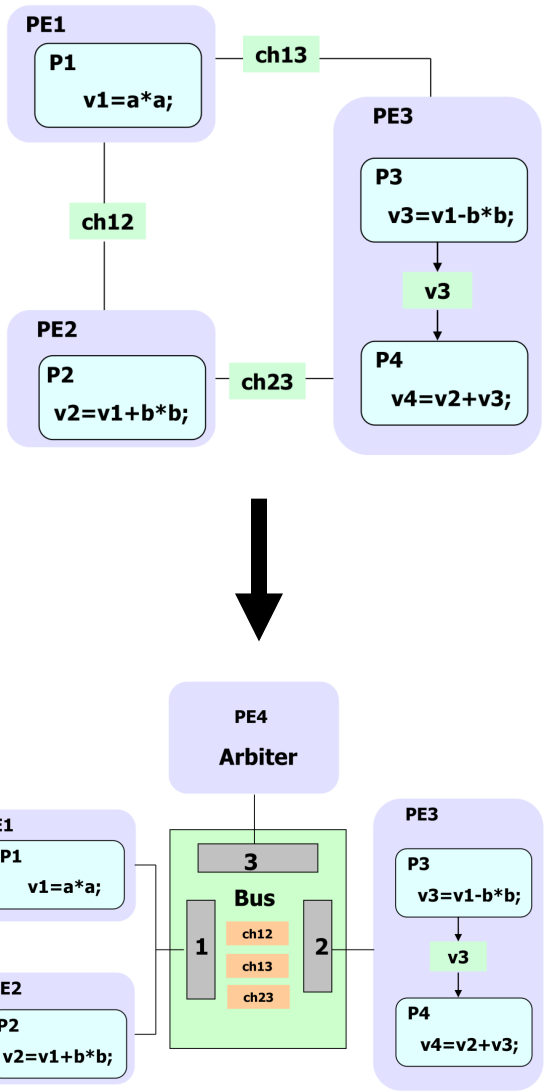
- Based on the analysis of the algorithm we need to:
  - partition the algorithm into Software/Hardware
  - select the general-purpose processor or the DSP
  - design IPs or select IPs from library
  - choose Real-Time Operation System (RTOS) if necessary

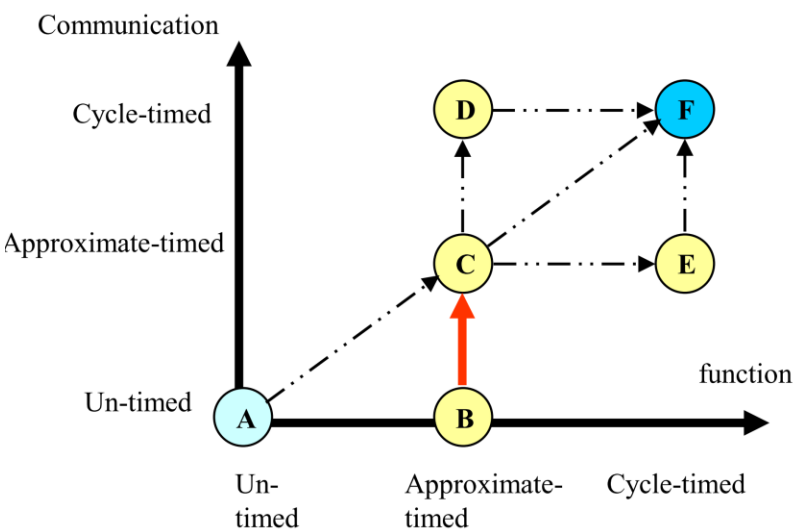




Communication Exploration

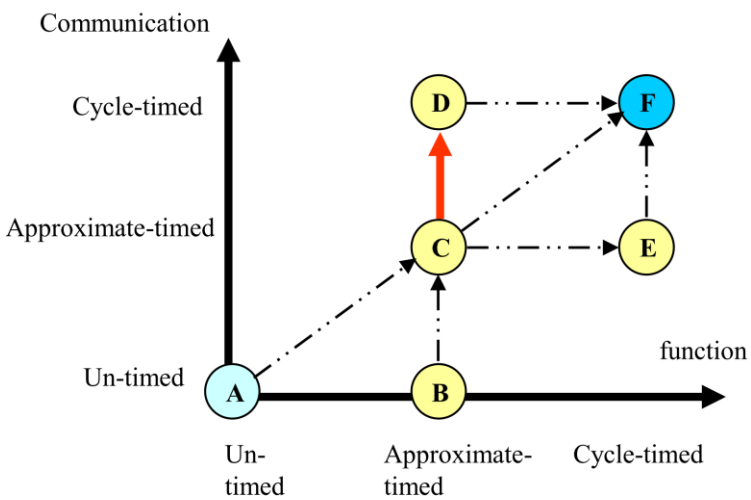
- We need to
  - map channels to buses (centralized or back-door)
  - assign bus-accessing properties for each IP (master or slave)
  - decide the bus arbitration policy

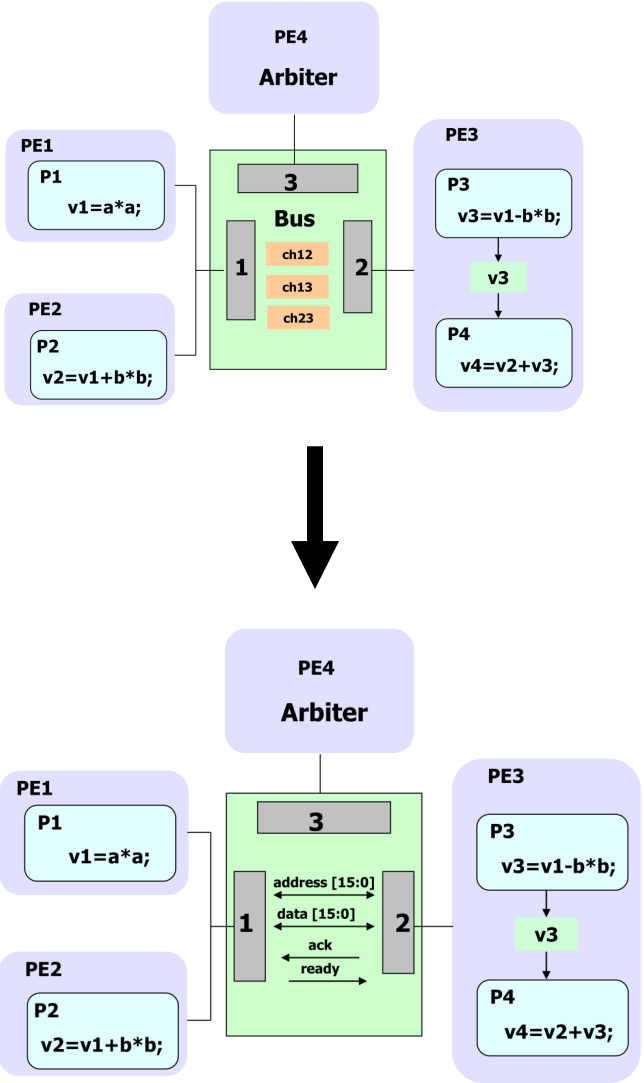




Protocol Refinement (Platform-based)

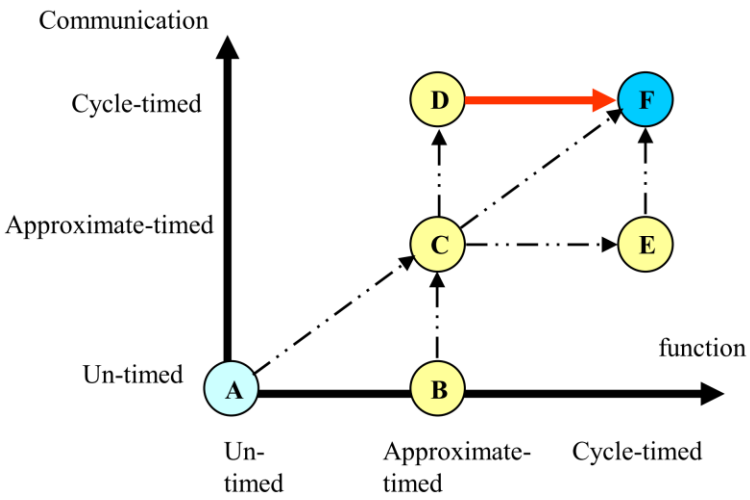
- We need to determine the pin- and cycle-accurate bus protocols.
- And the details of the bus control signal are contained.





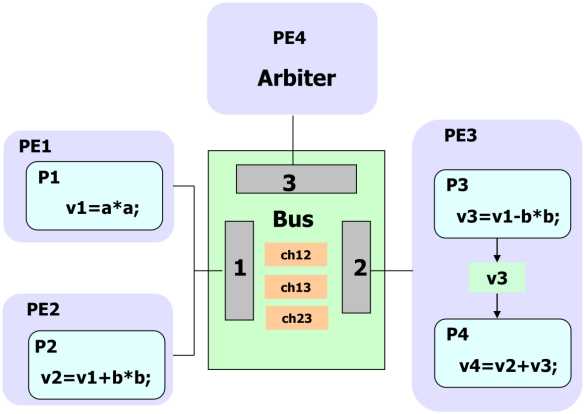
IP Refinement

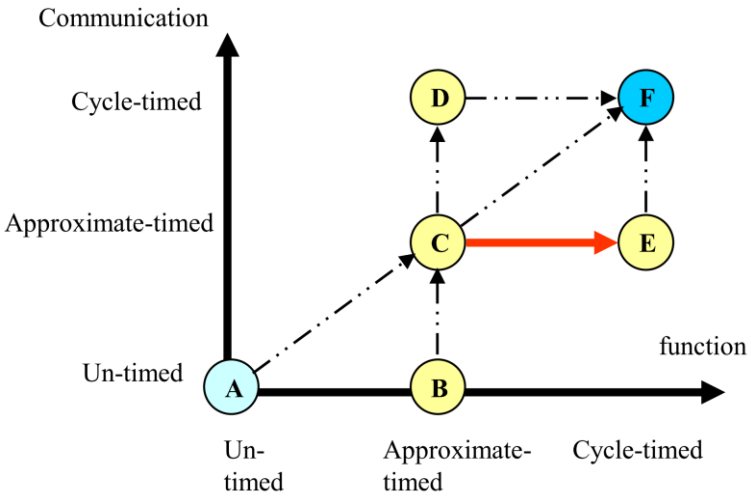
- The IPs are refined to pin-and cycle-accuracy.
- The embedded software is optimized to achieve high performance.
- The wrapper to transfer the data between IPs and bus are designed.

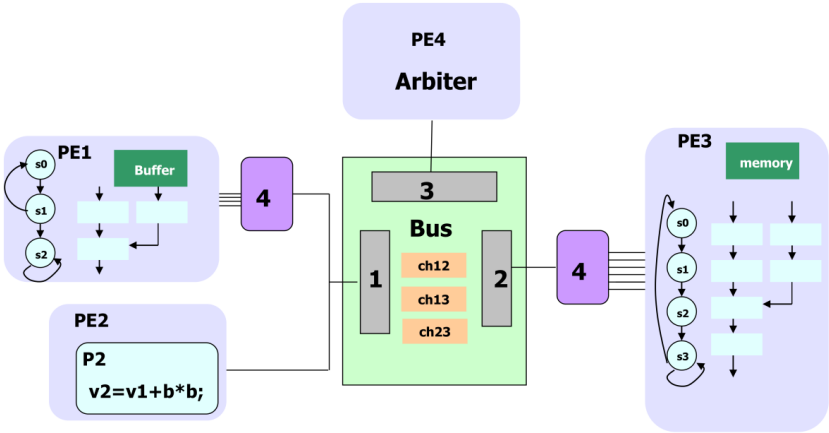


IP Replacement

- 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.

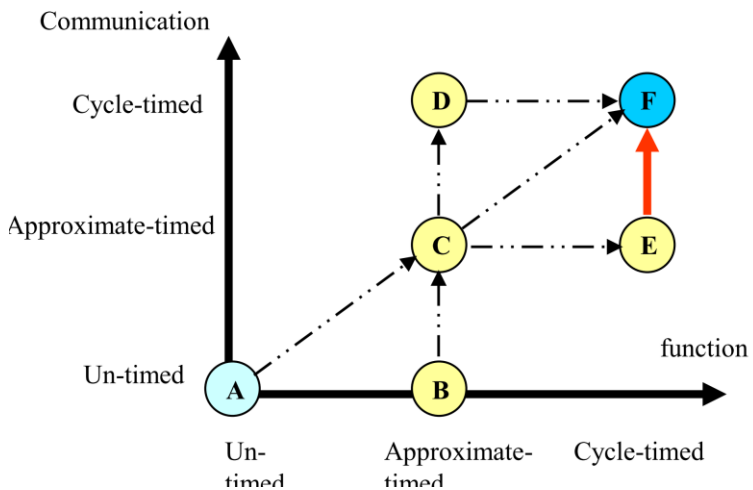




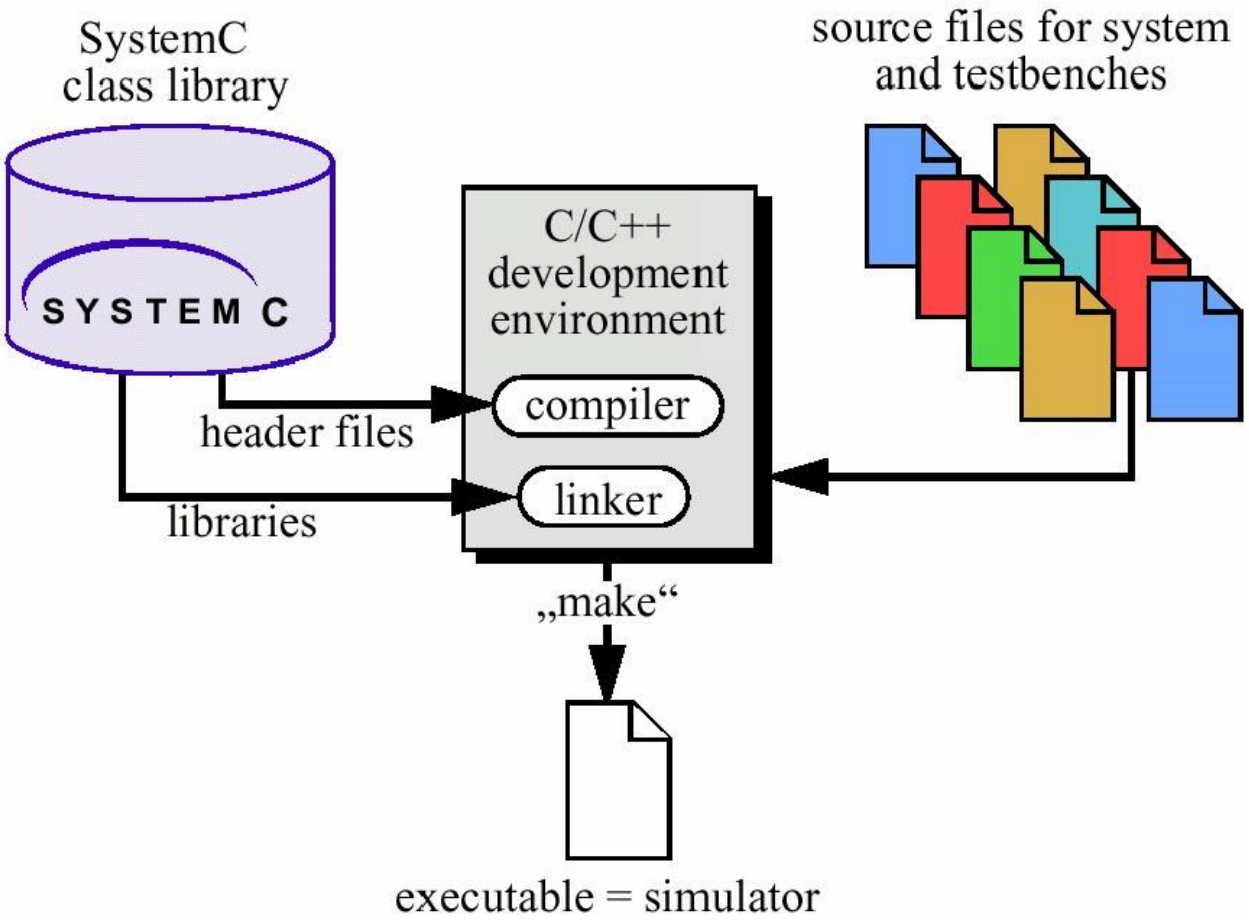


Communication Refinement

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

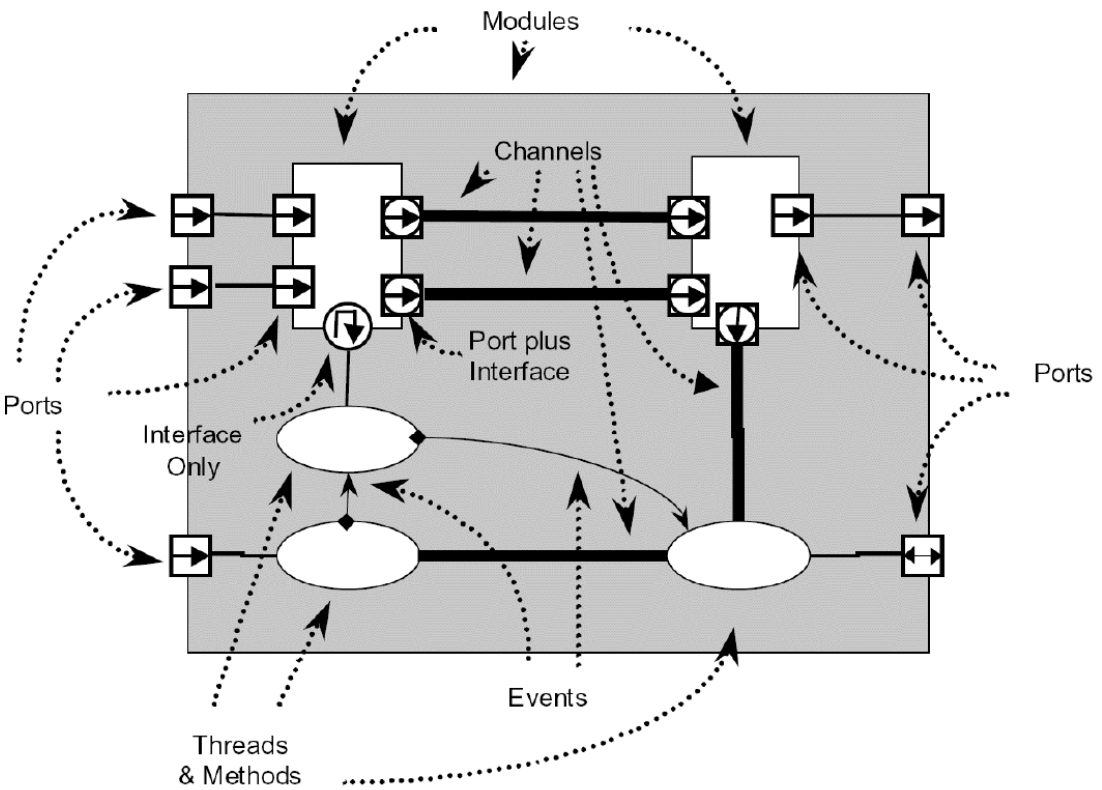


SystemC Design Flow



Important Concepts in SystemC

- Module
- Clock
- Process
  - Function
- Even-based simulation
- Communication protocols
  - Channel
  - Interface
- Concurrency



Event-based simulation

- In hardware design, the hardware will be activated when the input signal changes.
- In SystemC, the function is not called in order but is triggered according to the event
- An event is something that happens at specific point in time
  - Sensitive signal

Starting Pont: sc_main

- **C/C++**

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

- **SystemC**

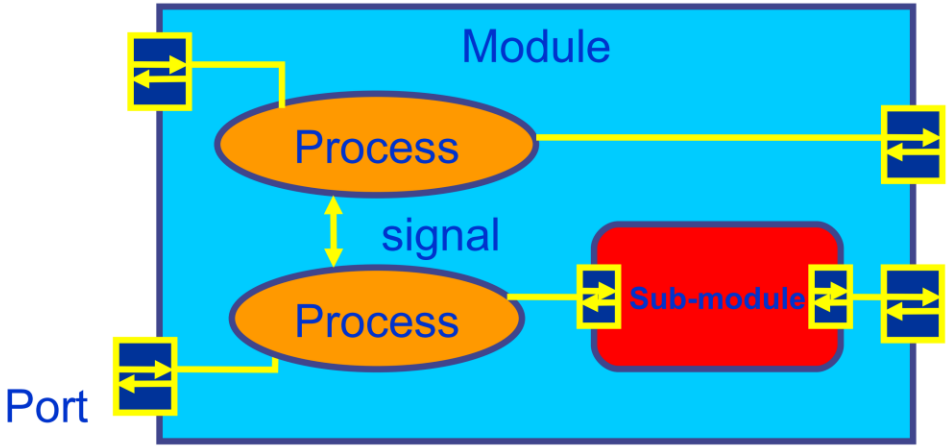
```
int sc_main(int argc, char* argv[])
{
    ELABORATION
    sc_start(); //Simulation begins & ends in this function
    [POST-PROCESSING]
    return EXIT_CODE; //Zero indicates success
}
```

- Must include “systemc.h”

Basic Unit of Design: SC_MODULE

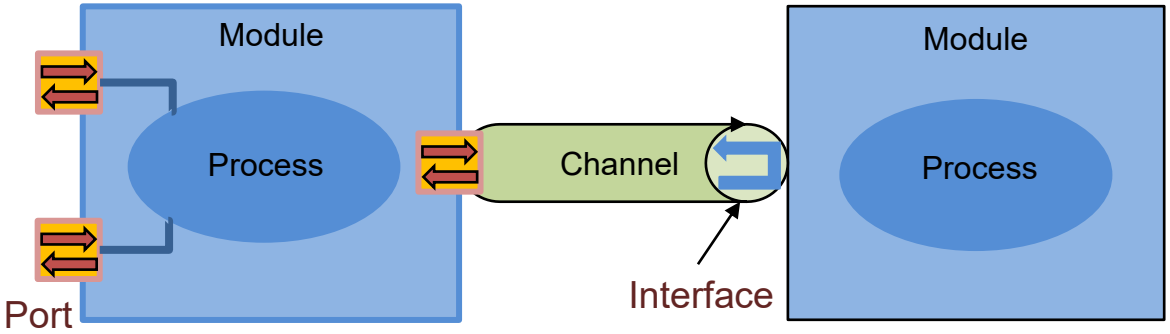
- A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity

- **Keyword: SC_MODULE**
- **Consist of**
  - Ports: communicate with outside (another channel)
  - Process: describe the functionality of module
  - Internal data and channels (sc_signal, …etc.)
  - Other modules (sub-module)



Communication

- When a block transfers data to another block, an interface for communication between blocks is required
  - Port
  - Channel
  - Interface



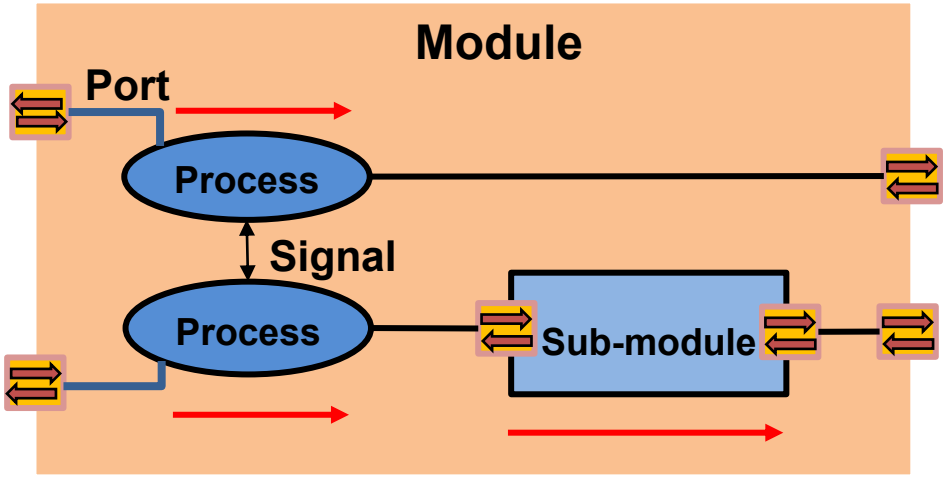
Parallel Processing

- C language execute the instructions sequentially
  - The function is called line by line in order

- Different from C language, the hardware design works in parallel
  - Each block operate at the same time

```
int main(int argc, char* argv[]){
        statement1
        statement2
        statement3
        statement4
        ...
        return EXIT CODE; //Zero indicates success
}
```

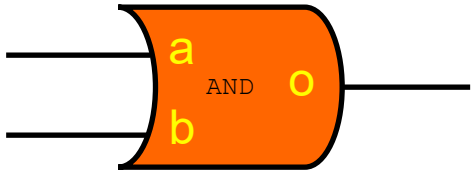
C/C++ Language



SystemC Language

A 2-input and-gate class in SystemC

This include file contains all systemc functions and base classes.



Instantiates the input pins `a` and `b`. They carry boolean signals.
This object inherits all systemC properties of a pin. how this is actually implemented is hidden from us!

All systemC classes start with sc_

This sets up a class containing a module with a functionality.

```
#include <systemc.h>

SC_MODULE(AND2)
{
  sc_in<bool> a;  // input pin a
  sc_in<bool> b;  // input pin b

  sc_out<bool> o; // output pin o

  SC_CTOR(AND2)     // the ctor
    {
      SC_METHOD(and_process);
      sensitive << a << b;
    }

  void and_process() {
    o.write( a.read() && b.read() );
  }
};
```

Similarly, a boolean output pin called `o`

This stuff is executed during construction of an and object

Tells the simulator which function to run to evaluate the output pin

This is run to process the input pins.

Calls read and write member functions of pins.

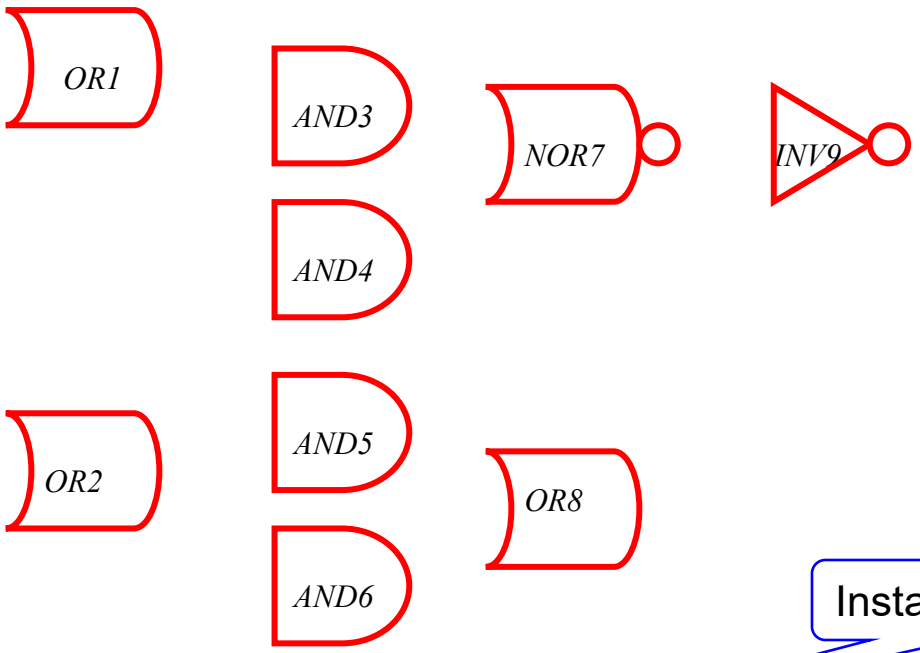
This is the actual AND operation!

SystemC Program Structure

```
#include <systemc.h>
#include "and.h"
#include "or.h"
// etc..

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

  // 1: Instantiate gate objects
  …
  // 2: Instantiate signal objects
  …
  // 3: Connect the gates to signals
  …

  // 4: specify which values to print

  // 5: put values on signal objects

  // 6: Start simulator run

}
```

- First a data structure is built that describes the circuit.
- This is a set of module (cell-) objects with attached pin objects.
- Signal objects tie the pins together.
- Then the simulation can be started.
- The simulation needs:
  - input values
  - the list of pins that is to reported.

Step 1: make the gate objects



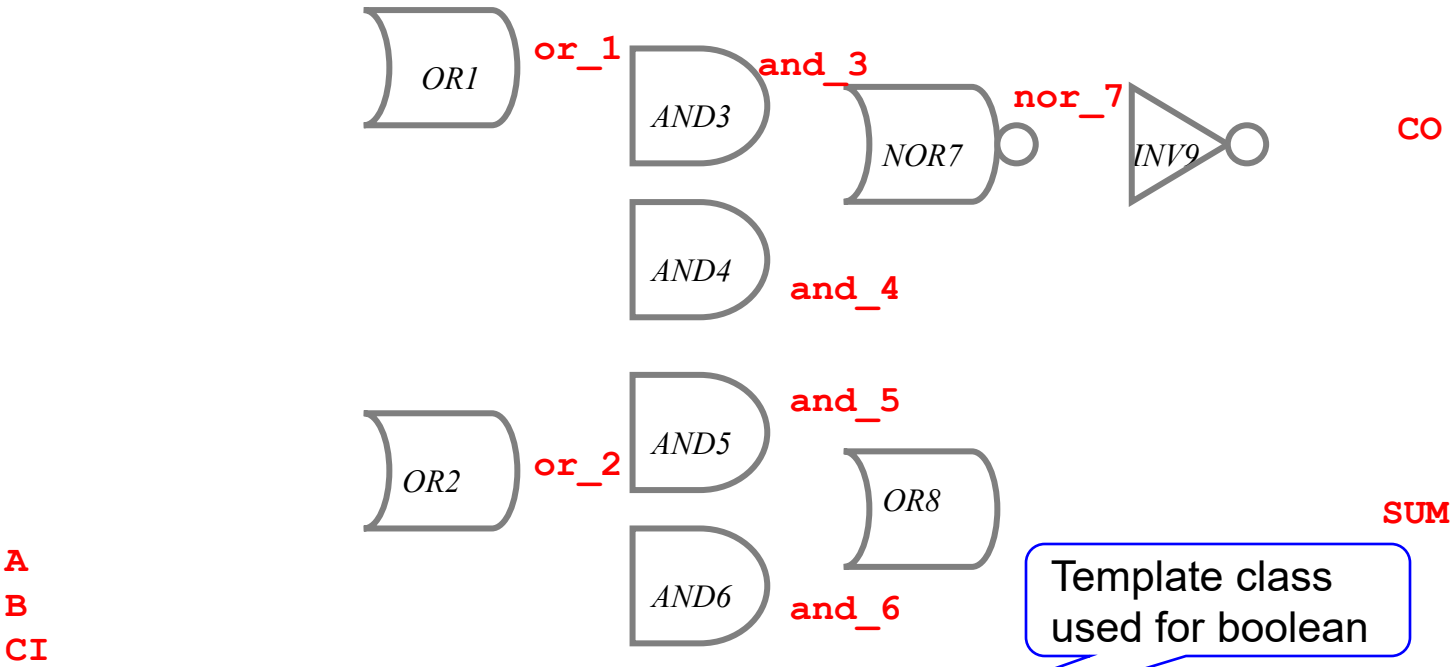
Instance name

Module type

```
// 1: instantiate the gate objects
OR2  or1("or1"), or8("or8");
OR3  or2("or2");
AND2 and3("and3"), and4("and4"), and5("and5");
AND3 and6("and6");
NOR2 nor7("nor7");
INV  inv9("inv9");

// … continued next page
```

Name stored in instance

Step 2: make the signal objects

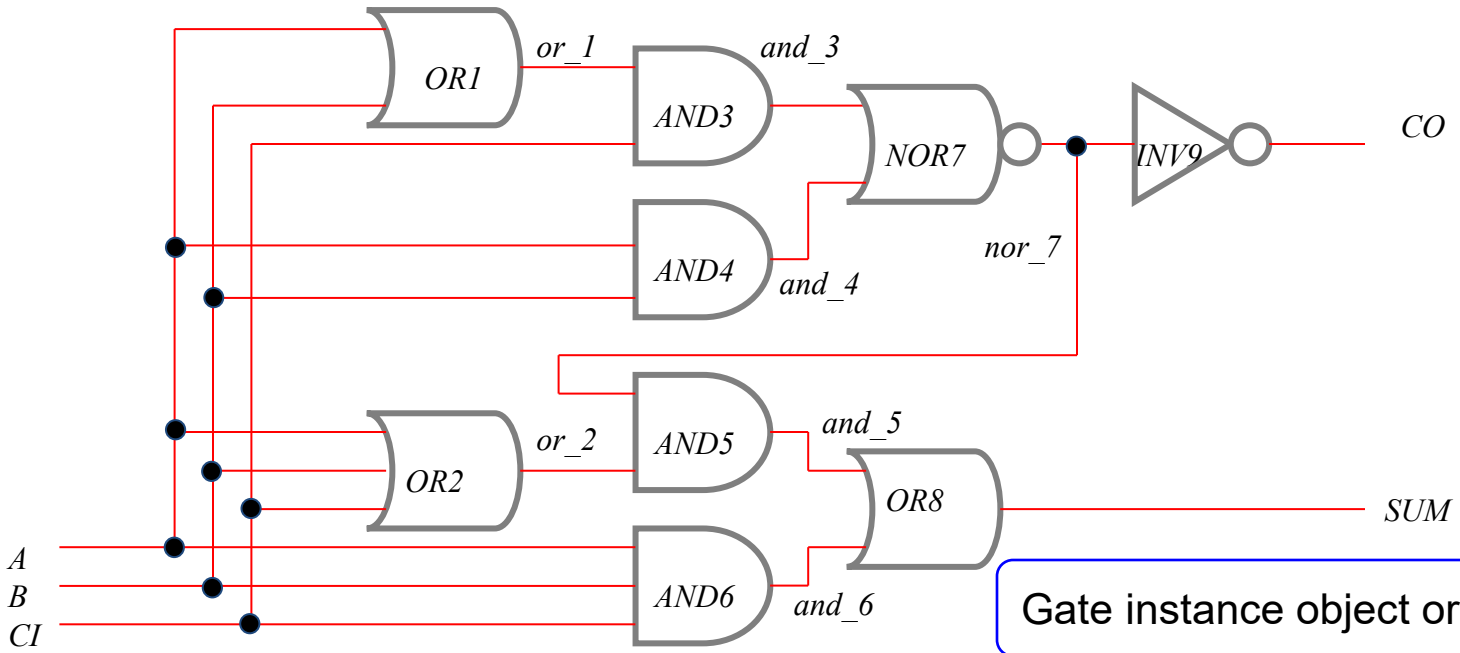


Template class used for boolean

Boolean signal

```
// … continued from previous page

// 2: instantiate the signal objects
sc_signal<bool> A, B, CI;        // input nets
sc_signal<bool> CO, SUM;         // output nets
sc_signal<bool> or_1, or_2, and_3, and_4;   // internal nets
sc_signal<bool> and_5, and_6, nor_7;        // internal nets

// … continued next page
```

Step 3: Connecting pins of gates to signals



Gate instance object or2

pin object o

Signal net object

```
// 3: Connect the gates to the signal nets
or1.a(A); or1.b(B); or1.o(or_1);
or2.a(A); or2.b(B); or2.c(CI); or2.o(or_2);
and3.a(or_1); and3.b(CI); and3.o(and_3);
and4.a(A); and4.b(B); and4.o(and_4);
and5.a(nor_7); and5.b(or_2); and5.o(and_5);
and6.a(A); and6.b(B); and6.c(CI); and6.o(and_6);
nor7.a(and_3); nor7.b(and_4); nor7.o(nor_7);
or8.a(and_5); or8.b(and_6); or8.o(SUM);
inv9.a(nor_7); inv9.o(CO);
// … continued next page
```

Running the simulation (in testbench)

```
  // .. continued from previous page
  sc_initialize();   // initialize the simulation engine

  // create the file to store simulation results
  sc_trace_file *tf = sc_create_vcd_trace_file("trace");

  // 4: specify the signals we'd like to record in the trace file
  sc_trace(tf, A, "A"); sc_trace(tf, B, "B"); sc_trace(tf, CI, "CI");
  sc_trace(tf, SUM, "SUM"); sc_trace(tf, CO, "CO");

  // 5: put values on the input signals
  A=0; B=0; CI=0;                  // initialize the input values
  sc_cycle(10);

  for( int i = 0 ; i < 8 ; i++ )  // generate all input combinations
    {
      A  = ((i & 0x1) != 0);       // value of A is the bit0 of i
      B  = ((i & 0x2) != 0);       // value of B is the bit1 of i
      CI = ((i & 0x4) != 0);       // value of CI is the bit2 of i
      sc_cycle(10);                // evaluate
    }

  sc_close_vcd_trace_file(tf);    // close file and we're done
}
```

Datatype

Overview

- SystemC provides the designer the ability to use any and all C++ data types as well as unique SystemC data types to model systems

Type	Description	Range	Precision
sc_bit	2 value single bit type	0,1	
sc_logic	4 value single bit type	0,1,Z,X	
sc_bv<W>	arbitrary sized 2 value vector type	0,1	
sc_lv<W>	arbitrary sized 4 value vector type	0,1,Z,X	
sc_int<W>	1~64 bit signed integer type	$-2^{W-1} \sim 2^{W-1}-1$	1
sc_uint<W>	1~64 bit unsigned integer type	$0 \sim 2^{W}-1$	1
sc_bigint<W>	arbitrary sized signed integer type	$-2^{W-1} \sim 2^{W-1}-1$	1
sc_biguint<W>	arbitrary sized unsigned integer type	$0 \sim 2^{W}-1$	1
sc_fixed<w,iw,q,o,n>	templated signed fix point type	$-2^{iw-1} \sim 2^{iw-1}-2^{iw-w}$	2^{iw-w}
sc_ufixed<w,iw,q,o,n>	templated unsigned fix point type	$0 \sim 2^{iw}-2^{iw-w}$	2^{iw-w}
sc_fix(OPs)	untemplated signed fix point type	$-2^{iw-1} \sim 2^{iw-1}-2^{iw-w}$	2^{iw-w}
sc_ufix(OPs)	untemplated unsigned fix point type	$0 \sim 2^{iw}-2^{iw-w}$	2^{iw-w}

sc_bit

- Two valued data type representing a single bit
- Can have the value ’0’(false) or ’1’(true) only
- Useful for modeling parts of the design where ‘Z’ (high impedance) or ‘X’ (unknown) values are not needed

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

Example: sc_bit

```
sc_bit a, b, c;          //declaration
bool d;

a = '0';                 //use char literals
b = '1';

a = a & b;
a &= b;                  //equivalent

d = true;
c = a | d;               //can be mixed with C/C++ bool type
```

sc_logic

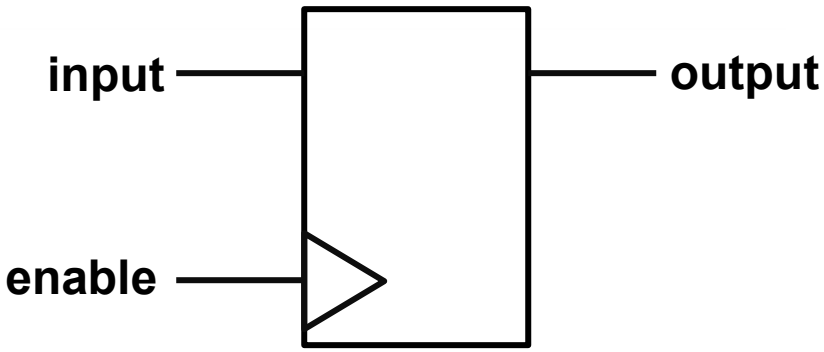
- A more general single bit data type
- Can have the value ’0’(false) , ’1’(true) , ‘Z’ (high impedance) or ‘X’ (unknown)
- Most common data type used for simulations at the RTL

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

Example: sc_logic

```
bool w;
sc_bit x;
sc_logic y, z;

w = x == y;                    //sc_bit and sc_logic
w = y != z;                    //sc_logic and sc_logic
w = y == '1';                  //sc_logic and literal

y = x;                         //sc_bit to sc_logic
x = y;                         //sc_logic to sc_bit

/*if the value of y is 'Z' or 'X' when assignment occurs, the result of
the assignment is undefined and a runtime warning is issued*/
```

Example: 3-state buffer




```
SC_MODULE(tristate_buf){
    sc_in< sc_bit > input;
    sc_out< sc_logic > output;
    sc_in< sc_bit > enable;

     void process() {
          sc_bit in, en;
          sc_logic out;

          in=input; en=enable;  // reading inputs to temporary variables
          if(en)
             out = in;
          else
             out = 'z';
          output = out;      // writing a temporary variable to output
    }

    SC_CTOR(tristate_buf) {
          SC_METHOD(process);
          sensitive<<enable<<input;
    }
};
```

sc_bv<W>

- Two valued arbitrary length vector to be used for large bit vector manipulation

Classes	Operators					
Bitwise	~	&	|	^	<<	>>
Assignment	=	&=	|=	^=		
Equality	==	!=				
Bit selection	[]					
Part selection	range()					
Concatenation	(,)					
Reduction	and_reduce()	or_reduce()	xor_reduce()			

Example: sc_bv<W>

```
sc_bit w;
sc_bv<8> x,y;
sc_bv<16> z;

z = “1111111111111111”;        //string of 0 and 1 can be assigned to sc_bv
y = “00000000”

w = z.or_reduce();             //or all bits in z together

w = z[4];                      //single bit selection

x = z.range(7,0)               //part selection
```

Example: sc_bv<W>

```
sc_bit w;
sc_bv<8> x, y;
sc_bv<16> z;

z = (x, y);                                  //concatenation of bit vectors

y = (z.range(15,12), z.range(3,0)); //mixed concatenation and part selection

(x.range(4,1), y.range(6,3)) = z.range(11,4);
/*bit selection, part selection and concatenation work on both sides of an
assignment operator and in expressions*/

cout<<"z = "<<z.to_string();        //print a human readable character string
```

sc_lv<W>

- Similar to sc_bv but can have ‘Z’ and ‘X’ value
- Always slower than sc_bv type in simulation

Classes	Operators					
Bitwise	~	&	|	^	<<	>>
Assignment	=	&=	|=	^=		
Equality	==	!=				
Bit selection	[]					
Part selection	range()					
Concatenation	(,)					
Reduction	and_reduce()	or_reduce()	xor_reduce()			

Example: sc_lv<W>

```
sc_lv<8> bus1;
if(enable)
  bus1 = "01xz10xx";
else
  bus1 = "zzzzzzzz";
cout<<bus1.to_string(); // OR: cout<<bus1
```

sc_int<W> , sc_uint<W>

- Fixed Precision Unsigned and Signed Integers
  - Integer variables with fixed width (number of bits)
    - Provides up to 64 bits (size of long int in C/C++ language)
  - Signed representation uses 2's complement

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

Example: sc_int<W> , sc_uint<W>

```
int w;
sc_bit b;
sc_int<16> x, y;
sc_uint<24> z;

w = 200;

z = w * 2;                    //sc_int and sc_uint can be used with int
                              //bits that exceed 24 are removed
                              //Z =0000 0000 0000 0001 1001 0000
b = z[3];                     //b =0
x = z.range(19, 4) ;          //selection operations work on sc_int or sc_uint
                              //x = 0000 0000 0001 1001
```

Example: sc_int<W> , sc_uint<W>

```
y = x<<4;                          //y = 0000 0001 1001 0000
z.range(15, 0) =  x & y;           //bitwise operations
                                   //z = 0000 0000 0000 0000 0001 0000

y += x.range(7, 4);                //y = 0000 0001 1001 0001
x++;
//perform compound-assignment and autoincrement like int

z = (x.range(7, 0), y);            //y = 0001 1010 0000 0001 1001 0001
/*concatenation operation can be used to make a larger value from one or
more smaller values*/
```

sc_bigint<W> , sc_biguint<W>

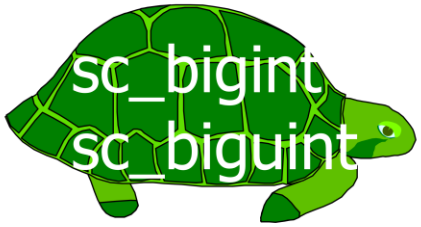
- Integer with (virtually) no width limit
  - Provides up to MAX_NBITS bits (defaulted to 512)
  - MAX_NBITS defined in sc_constants.h

- Signed representation uses 2's complement
- Operators are the same as sc_int, sc_uint

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

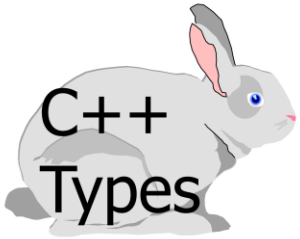
Speed Issue

- Integer Types

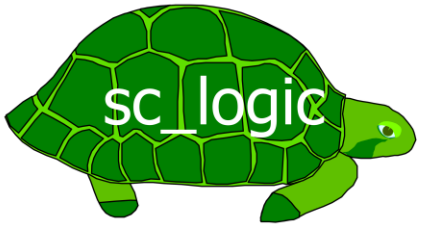


with
MAX_NBITS
not defined

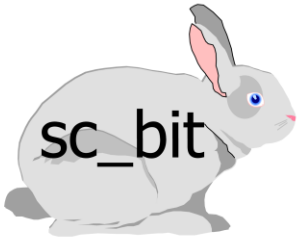
sc_int
sc_uint



- Bit and Logic Types



sc_lv



sc_bv

Fixed Point Types (1/4)

- When designers model at a high level, floating point numbers are useful to model arithmetic operations
  - can handle a very large range of values
  - easily scaled
- In hardware floating point data types are typically converted or built as fixed-point data types
  - to minimize the amount of hardware needed to implement the functionality

Fixed Point Types (2/4)

- 4 basic fixed-point type in SystemC
- sc_fixed
- sc_fix
- sc_ufixed
- sc_ufix
- sc_fixed and sc_ufixed uses static arguments to specify the functionality
  - setup at compile time and do not change
- sc_fix and sc_ufix can use argument types that are nonstatic
  - sc_fix and sc_ufix can use variables to determine arguments at runtime (i.e., even-driven datatype adjustment)

Fixed Point Types (3/4)

- No mixing of signed / unsigned / other type is allowed
- When perform binary bitwise operations, two operands are aligned by the binary point (.)
  - the maximum word length and maximum fractional word length are taken
  - Both operands are converted to this type first

Classes	Operators								
Bitwise	~	&	|	^					
Arithmetic	+	-	*	/	%	<<	>>		
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Autoincrement	++								
Autodecrement	--								
Bit selection	[] (return sc_fxnum_bitref)								
Part selection	range() (return sc_fxnum_subref)								

Fixed Point Types (4/4)

- Fixed point type is declared with the following syntax:

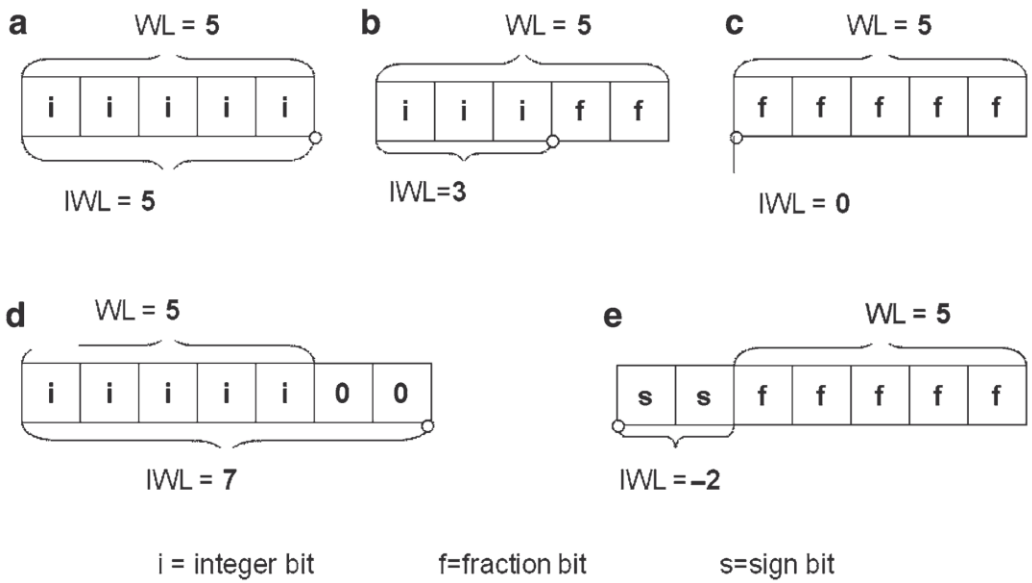
- sc_fixed<wl, iwl, q_mode, o_mode, n_bits> x;
- sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> y;

Focus in this class

- sc_fix x(list of options);
- sc_ufix y(list of options);

arguments for sc_fixed and sc_ufixed (1/2)

- wl - Total word length
  - the total number of bits used in the type
- iwl - Integer word length
  - the number of bits that are to the left of the binary point (.)



arguments for sc_fixed and sc_ufixed (2/2)

- q_mode - Quantization mode
  - determines the behavior of the fixed point type when the result of an operation generates more precision in the least significant bits than is available as specified by the word length and integer word length parameters
- o_mode - Overflow mode
  - determines the behavior of the fixed point most significant bits when an operation generates more precision in the most significant bits than is available
- n_bits - Number of saturated bits
  - only used for overflow mode and specifies how many bits will be saturated if a saturation behavior is specified and an overflow occurs

wl and iwl

- wl
  - must be greater than 0
- iwl
  - can be positive or negative, and larger than the word length

wl	iwl	representation	signed range	unsigned range
5	7	xxxxx00.	[-64,60]	[0,124]
5	5	xxxxx.	[-16,15]	[0,31]
5	3	xxx.xx	[-4,3.75]	[0,7.75]
5	1	x.xxxx	[-1,0.9375]	[0,1.9375]
5	0	0.xxxxx	[-0.5,0.46875]	[0,0.96875]
5	-2	0.ssxxxxx	[-0.125,0.109375]	[0,0.234375]
1	-1	0.sx	[-0.25,0]	[0,0.25]

Quantization Mode

- Determine what happens to the LSBs when more bits of precision are required than are available

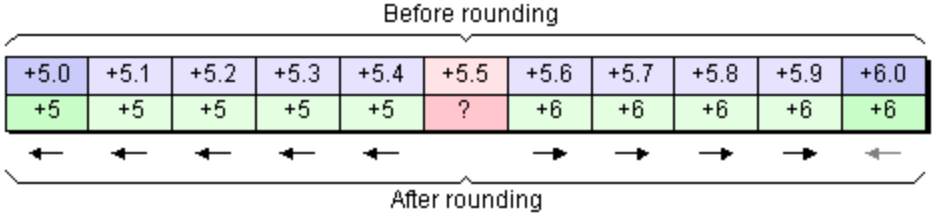
- EX : 1011.011 assigned to a sc_fixed in format of xxxx.xx

1011.01~~1~~

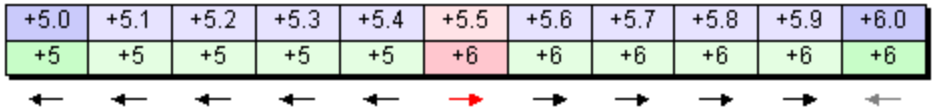
Quantization Mode	Name
Round	SC_RND
Round towards zero	SC_RND_ZERO
Round towards minus infinity	SC_RND_MIN_INF
Round towards infinity	SC_RND_INF
Convergent rounding	SC_RND_CONV
Truncate (default)	SC_TRN
Truncation toward zero	SC_TRN_ZERO

SC_RND

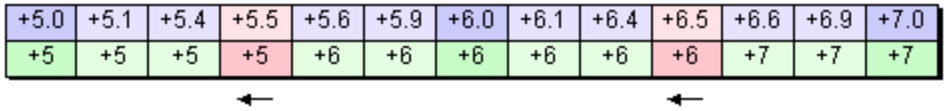
- Round the value to the closest representable number
- Accomplished by adding the MSB of the removed bits to the remaining bits
- 1011.011 → 101101 + 1 → 101110 → 1011.10



Round-Toward-Nearest



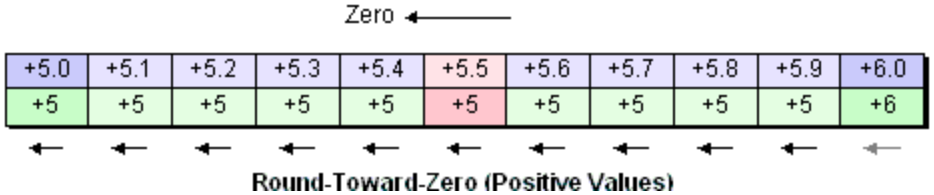
Round-Half-Up (Positive Values)

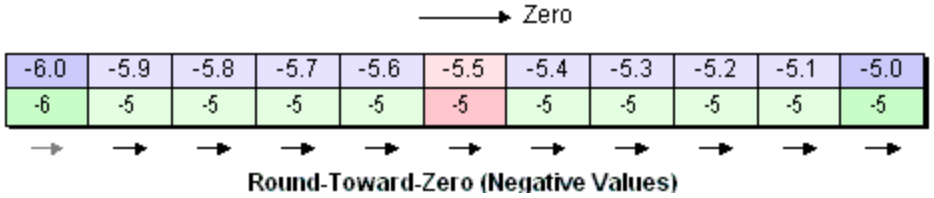


Round-Half-Down (Odd and Even Positive Values)

SC_RND_ZERO

- Perform SC_RND if the two nearest representable numbers are not equal distance apart
- Otherwise rounding to zero will be performed
- For positive numbers the redundant bits are simply deleted, for negative numbers the MSB of the deleted bits is added to the remaining bits
- 1011.011 → 101101 + 1 → 101110 → 1011.10
- 0011.011 → 001101 - 1 → 001100 → 0011.00

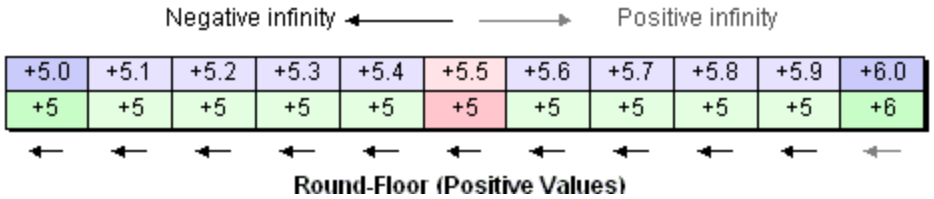


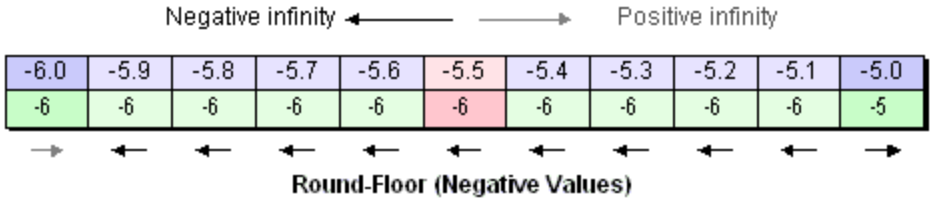


SC_RND_MIN_INF

- Perform SC_RND if the two nearest representable numbers are not equal distance apart
- Otherwise round towards minus infinity by eliminating the redundant bits

- 1011.011 → 1011.01





SC_RND_INF

- Perform SC_RND if the two nearest representable numbers are not equal distance apart
- Otherwise the number is rounded towards plus infinity if positive or minus infinity if negative
- For positive numbers the MSB of the deleted bits is added to the remaining bits, for negative numbers the redundant bits are deleted
- 1011.011 → 1011.01

Negative infinity ← → Positive infinity

+5.0	+5.1	+5.2	+5.3	+5.4	+5.5	+5.6	+5.7	+5.8	+5.9	+6.0
+5	+6	+6	+6	+6	+6	+6	+6	+6	+6	+6

Round-Ceiling (Positive Values)

Negative infinity ← → Positive infinity

-6.0	-5.9	-5.8	-5.7	-5.6	-5.5	-5.4	-5.3	-5.2	-5.1	-5.0
-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-5

Round-Floor (Negative Values)

SC_RND_CONV

- Perform SC_RND if the two nearest representable numbers are not equal distance apart

- Otherwise this mode checks the LSB of the remaining bits, if the LSB is 1 this mode will round towards plus infinity, if the LSB is 0 this mode will round towards minus infinity

- 1011.011 → 101101 + 1
  → 101110 → 1011.10

- 1011.010 → 101101 + 0
  → 101101 → 1011.01

SC_TRN

- The **default quantization mode** to be used if no other value is specified
- The result is always rounded towards minus infinity
- The redundant bits are always deleted no matter whether the number is positive or negative
- 1011.011 → 1011.01

SC_TRN_ZERO

- Perform SC_RND for positive numbers
- For negative numbers the result is rounded towards zero (SC_RND_ZERO)
- Accomplished by deleting the redundant bits on the right side and adding the sign bit to the LSBs of the remaining bits
- Only occurs if at least one of the deleted bits is nonzero
- 1011.01~~1~~ → 101101 + 1 → 101110 → 1011.10

Saturation Mode

- Determine what happens when the result of an operation generates more bits on the MSB side than are available

- EX : 1011.011 assigned to a sc_fixed in format of xxx.xxx

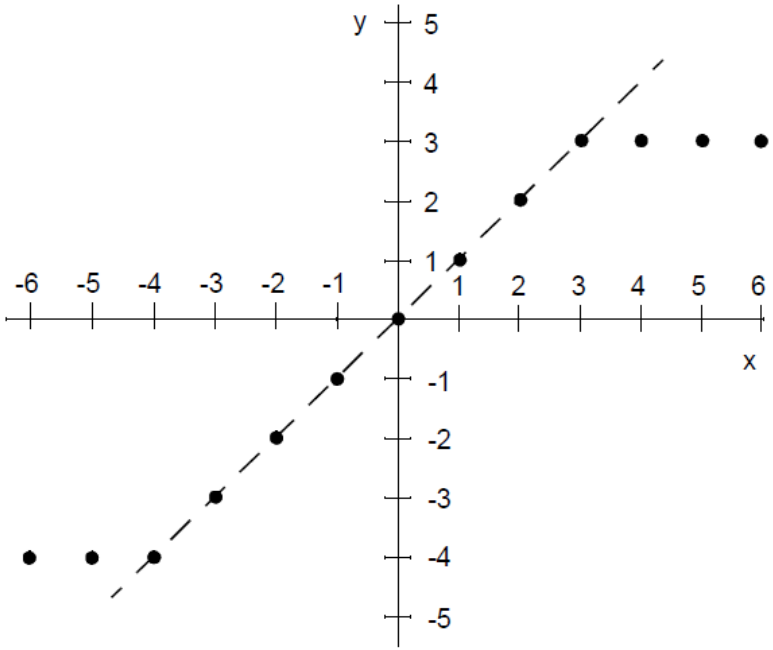
1011.011

Saturation Mode		Name
Saturation		SC_SAT
Saturation to zero		SC_SAT_ZERO
Symmetrical saturation		SC_SAT_SYM
Wrap-around	n_bits = 0 (default)	SC_WRAP
	n_bits > 0	
Sign magnitude wrap-around	n_bits = 0	SC_WRAP_SM
	n_bits > 0	

SC_SAT

- Convert the specified value to MAX for overflow or MIN for an underflow

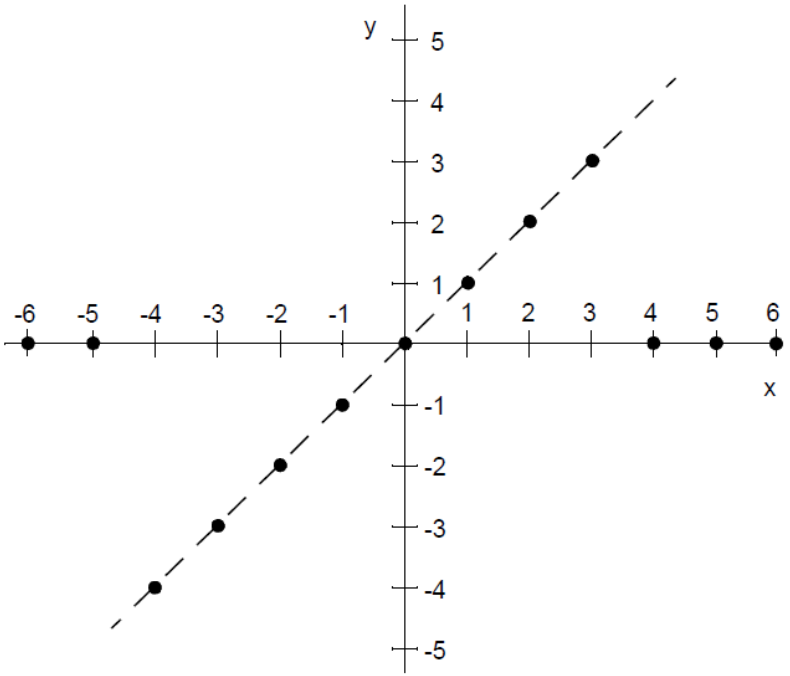
- 1011.011 → 1 011.011 → 100.000



SC_SAT_ZERO

- Set the result to 0 for any input value that is outside the representable range

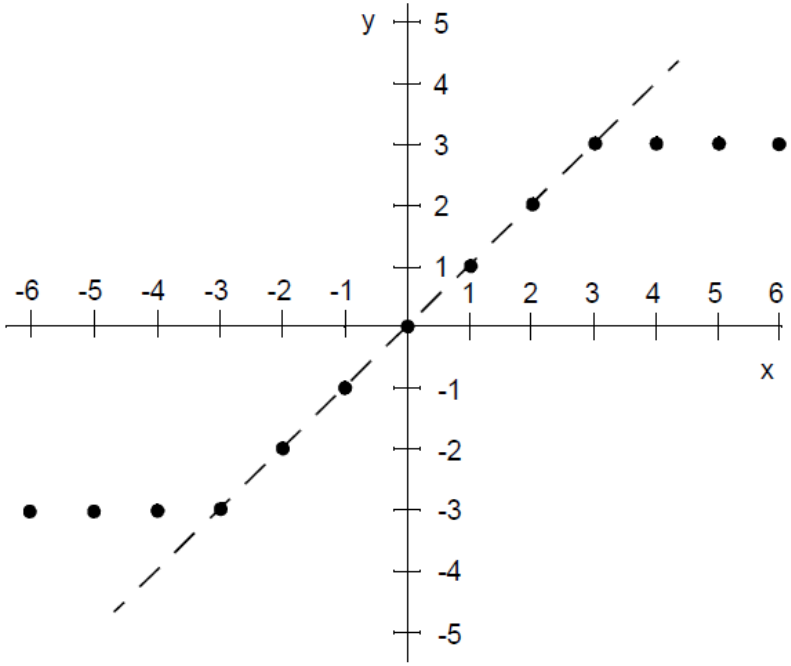
- 1011.011 → 1 011.011 → 000.000



SC_SAT_SYM

- Convert the specified value, defined by designer, to MAX for overflow or -MAX for an underflow

- 1011.011 → 1 011.011 → 101.000



What is the difference between SC_SAT and SC_SAT_SYM?

SC_WRAP , n_bits = 0

- The default overflow mode
- Any MSB bits outside the range of the target type are deleted
- 1011.011 → 1 011.011 → 011.011

SC_WRAP , n_bits > 0

- n_bit MSB bits are to be saturated
- The sign bit is retained so that positive numbers remain positive and negative numbers remain negative
- The bits that are not saturated are simply copied

For n_bits = 2

1011.011 → 1 011.011 → 101.011

SC_WRAP_SM , n_bits = 0

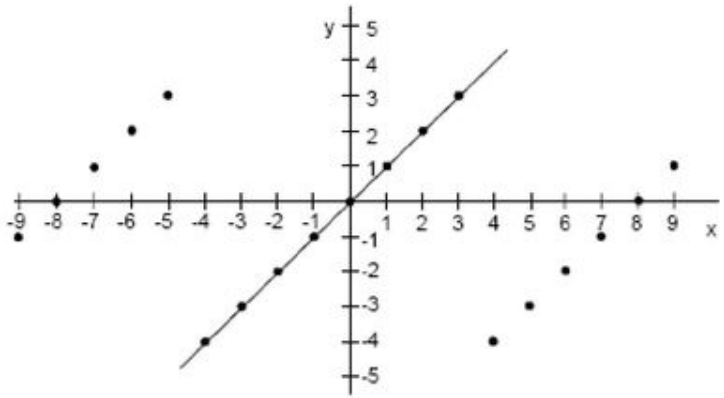
- Delete any MSB bits that are outside the result word length
- If the MSB of remaining bit is different from the LSB of deleted bits, all the remaining bits are inverted
- 1011.011 → 1 011.011 → 100.100

SC_WRAP_SM , n_bits > 0

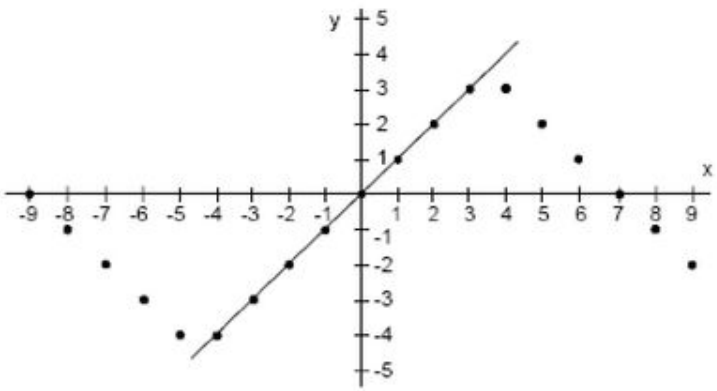
- Delete any MSB bits that are outside the result word length
- n_bits MSB bits will be saturated and the sign bit retained
- If the LSB of saturation bits is different from the original bit, the remaining bits are inverted
- For n_bits = 2
- 1011.011 → 1 011.011

  → 101.011 → 100.100

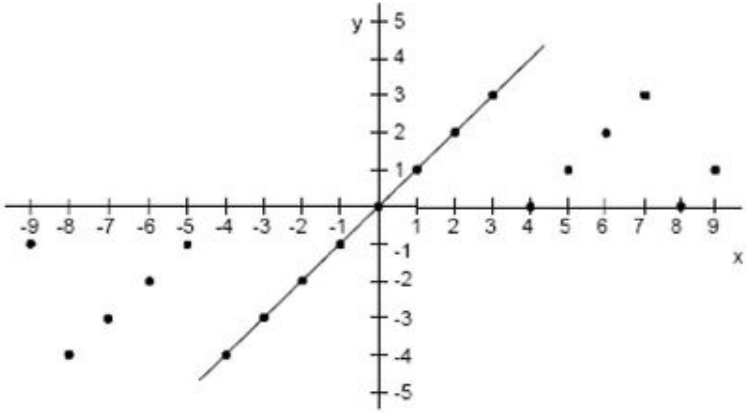
Others



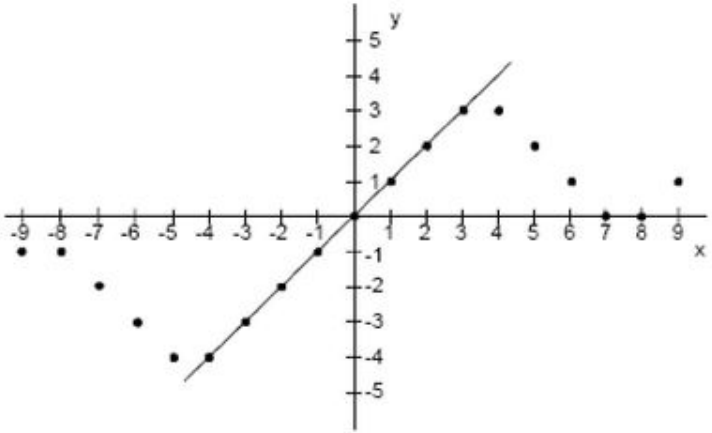
SC_WRAP, n_bits=0



SC_WRAP_SM, n_bits=0



SC_WRAP, n_bits=1



SC_WRAP_SM, n_bits=1

Performance issue

- Datatypes that close to C++ built-in types would be faster

	Datatypes			
Fastest	C++ built-in types (e.g. int, char and bool)			
	sc_int<>	sc_uint<>		
	sc_bit	sc_bv<>		
	sc_logic	sc_lv<>		
	sc_bigint<>	sc_biguint<>		
Slowest	sc_fixed<>	sc_fix	sc_ufixed<>	sc_ufix