ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN ĐIỆN TỬ
----------o0o----------

LUẬN VĂN TỐT NGHIỆP ĐẠI HỌC

THIẾT KẾ HỆ THỐNG SYSTEM ON CHIP (SoC) BẰNG NGÔN NGỮ C++ VÀ SYSTEMC

GVHD: **ThS PHẠM ĐĂNG LÂM**
SVTH: **PHẠM GIA HUY**
MSSV: **41201386**

TP. HỒ CHÍ MINH, THÁNG 12 NĂM 2017

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
-----☆-----

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập – Tự do – Hạnh phúc.
-----☆-----

Số: _____ /BKĐT
Khoa: **Điện – Điện tử**
Bộ Môn: **Điện Tử**

NHIỆM VỤ LUẬN VĂN TỐT NGHIỆP

1. HỌ VÀ TÊN: MSSV:

2. NGÀNH: **ĐIỆN TỬ - VIỄN THÔNG** LỚP :

3. Đề tài:

4. Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):

.....
.....
.....
.....
.....
.....

5. Ngày giao nhiệm vụ luận văn: .....

6. Ngày hoàn thành nhiệm vụ: .....

7. Họ và tên người hướng dẫn: Phần hướng dẫn

....
....

Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

Tp.HCM, ngày….. tháng….. năm 20

CHỦ NHIỆM BỘ MÔN **NGƯỜI HƯỚNG DẪN CHÍNH**

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):.....
Đơn vị:.....
Ngày bảo vệ : .....
Điểm tổng kết: .....
Nơi lưu trữ luận văn: .....

LỜI CẢM ƠN

Lời đầu tiên em chân thành cảm ơn sự hướng dẫn nhiệt tình, đóng góp ý kiến quý báu của thầy Phạm Đăng Lâm cùng với sự hỗ trợ của bộ môn Kỹ Thuật Điện Tử, khoa Điện – Điện Tử, Trường Đại Học Bách Khoa Thành Phố Hồ Chí Minh.

Em vô cùng biết ơn gia đình luôn ủng hộ và tạo điều kiện tốt nhất để em học tập rèn luyện trong suốt hơn 4 năm học vừa qua.

Luận văn đại học được xem như cánh cửa mở ra cho sinh viên con đường bước vào môi trường làm việc chuyên nghiệp với hành trang là những tri thức, kinh nghiệm học được từ thầy cô và bạn bè. Thực hiện đề tài luận văn đã cho em nhiều kiến thức mới, tổng hợp, vận dụng được những điều đã học, đặc biệt là việc thiết kế hệ thống System On Chip (SoC) bằng ngôn ngữ C++ và SystemC.

Tp. Hồ Chí Minh, ngày 20 tháng 12 năm 2017 .

Sinh viên

TÓM TẮT LUẬN VĂN

Luận văn này trình bày về …

MỤC LỤC

DANH SÁCH HÌNH MINH HỌA

DANH SÁCH BẢNG SỐ LIỆU

1. GIỚI THIỆU

2. KIẾN THỨC CƠ BẢN VỀ SYSTEMC

2.1. Kiểu dữ liệu

Chương này giúp chúng ta hình dung các kiểu dữ liệu mà SystemC hỗ trợ. Việc sử dụng kiểu dữ liệu nào cho trường hợp nào hay chuyển thể giữa các kiểu dữ liệu cần thời gian và project thực hiện

2.1.1. Biểu diễn số

C++ cho phép biểu diễn các kiểu đơn giản như: **integers, floats, Booleans, characters** và **strings.** Trong đó kiểu **sc_string** là dạng tổng quát để miêu tả kiểu dữ liệu và cũng là dạng trung gian để chuyển giữa các kiểu dữ liệu. Đôi khi một số kiểu dữ liệu (Bản chất là class) có hổ trợ hàm để chuyển kiểu của nó sang một kiểu khác. Đôi khi kiểu dữ liệu đó không có hổ trợ. Trong lớp sc_string, các hàm chuyển được hổ trợ để có thể chuyển đổi kiểu dữ liệu dễ dàng.

Ví du:

sc_int *<6>* *// có 5 bit giá trị và 1 bit dấu*

sc_uint*<6>* *// có 6 bit giá trị*

sc_fixed*<6,2>* *// 2 bit cho phần lẻ và 4 bit cho phần nguyên*

sc_string *name (“0* **base** *[sign]* **number** *[e[+|-]* **exp***]”);*

Trong đó:

- ***Base****: một trong các kiểu b (binary), o (octal), d (decimal) và x (hexadecimal)*

- ***Sign****: signed (để trống), unsigned (us), signed magnitude (sm) và cannical sign Digit (csd)*

-number*: là kiểu integer dựa theo base*

-exp*: Phần mũ - luôn sử dụng decimal*

Nếu không dùng kiểu sc_string vì quá rườm rà thì người dùng có thể sử dụng chính xác kiểu dữ liệu mong muốn sau đây

sc_numrep	***Prefix***	***Meaning***	***sc_int<5>(-13)^{10}***
SC_DEC	*0d*	*Decimal*	*-0d13*
SC_BIN	*0b*	*Binary*	*0b10011*
SC_BIN_US	*0bus*	*Binary unsigned*	*0bus01101*
SC_BIN_SM	*0bsm*	*Binary signed magnitude*	*-0bsm01101*
SC_OCT	*0o*	*Octal*	*0o03*
SC_OCT_US	*0ous*	*Octal unsigned*	*0ous15*
SC_OCT_SM	*0osm*	*Octal signed magnitude*	*-0osm03*
SC_HEX	*0x*	*Hex*	*0xf3*
SC_HEX_US	*0xus*	*Hex unsigned*	*0xus0d*
SC_HEX_SM	*0xsm*	*Hex signed magnitude*	*-0xsm0d*
SC_CSD	*0csd*	*Canonical signed digit*	*0csd-101-*

Ví dụ:

string a (“0d13”) //decimal 13

a= sc_string (“0b101110”) //decimal 44

2.1.2. Kiểu tự nhiên

SystemC hỗ trợ tất cả các kiểu tự nhiên của C++: **int, long int, unsigned int, unsigned long int, unsigned short int, short, double, float, char, bool.**

Gọi là kiểu tự nhiên vì đây là các kiểu dữ liệu mà C cũng hổ trợ, trình biên dịch có lịch sử lâu đời khi hổ trợ các kiểu dữ liệu này và đã tối ưu tài nguyên phần cứng khi khai báo/dùng các kiểu này. Do đó, khi lập trình với SC thì được khuyên dùng tốt nhất là các kiểu native nếu được lựa chọn

Ví dụ:

```
int             spark_offset    //điều khiển tia lửa
unsigned        repair=0        // count =0
unsigned long   mile            // đo mét
short int       speed           // tốc độ
float           temperature     // nhiệt độ
double          time            // thời gian
```

std::string license_plate // bản quyền

const bool warning_light=true trạng thái

enum compass {SW,W,NW,N,NE,E,SE,S}

2.1.3. Kiểu toán học

SystemC cung cấp 2 loại dữ liệu toán học: 64bit va lớn hơn 64bit.

sc_int và sc_uint

sc_int vas c_uint dùng trong trường hợp dữ liệu có độ rộng từ 1-64 bit.

Ví dụ:

sc_int <8> ex1

sc_uint<5> ex2

Chú ý: Bất kì kiểu dữ liệu nào không phải tự nhiên(native data type) thì sẽ làm cho phần cứng chạy chậm vì mất thời gian xử lý dữ liệu trước khi chuyển về kiểu dữ liệu native/dữ liệu bít.

sc_bigint và sc_biguint

Trong trường hợp số bit sử dụng lớn hơn 64bit, SystemC (SC) cung cấp kiểu **sc_bigint** và **sc_biguint**.

sc_bigint< BITWIDTH > NAME;

sc_biguint< BITWIDTH > NAME;

Ví dụ:

sc_int<5> seat_position=3; // 5bits: 4 bits tín hiệu, 1 bit dấu

sc_uint<13> days_SLOC(4000); // 13 bits không dấu

sc_biguint<80> revs_SLOC; // 80bits không dấu

Chú ý: Không sử dụng sc_bigint trong trường hợp ít hơn 64bits vì phần cứng hiện tại chủ yếu là 64 bít nên việc dùng dữ liệu lớn hơn 64 bít bắt trình biên dịch sẽ phải xử lý dữ liệu (cắt thành hai chuỗi 64 bít) trước khi chuyển đến xử lý ở cấp thấp hơn làm chương trình chạy chậm.

2.1.4. Kiểu luận lý (boolean) và kiểu multi-value data

2.1.4.1. sc_bit và sc_bv

SystemC cung cấp **sc_bit** cho giá trị 0 và 1, **sc_bv** cho trường hợp vector (một bus có nhiều bít). Những kiểu này thì cũng không hỗ trợ toán học giống như **sc_int** và cũng không thực thi nhanh như **bool** và **Standard Template Library** bitset.

sc_bit NAME;

sc_bv<BITWIDTH> NAME;

Ví dụ:

sc_bit exam1;

sc_bv<4> exam2;

sc_bit và **sc_bv** cũng hỗ trợ vài hằng số dữ liệu **SC_LOGIC_1** và **SC_LOGIC_0**. Để ít đánh máy hơn, nếu sử dụng "**namespace**" **sc_dt**, thì sau đó chỉ cần dùng **Log_1** và **Log_0**, hoặc là kiểu '**1**' và '**0**'. Khái niệm "**namespace**" chẳng qua là nơi chứa nhiều thư viện, thư viện chính là các lớp có sẵn, các lớp chính là nhiều biến và hàm nằm trong đó có mối quan hệ.

Các phép tính logic trên bít phổ biến được hổ trợ: **and, or, xor** (ký hiệu trong code: &, |, ^). Trong trường hợp muốn tính các phép toán logic trên các bít của một vector, một số hàm được hổ trợ ([] và **range()**), **sc_bv<>** cũng hỗ trợ **and_reduce(), or_reduce(), nand_reduce(), nor_reduce(), xor_reduce(), và xnor_reduce().**

Ví dụ:

sc_bit flag(SC_LOGIC_1);

sc_bv<5> position = "01101";

sc_bv<6> mask = "100111";

sc_bv<5> active = positions & mask;

sc_bv<1> all = active.and_reduce();

position.range (3,2) = "00";

position[2] = active[0] ^ flag;

2.1.4.2. sc_logic và sc_lv

Thú vị hơn kiểu Boolean và multi-value data, kiểu dữ liệu này mô tả được cả các trường hợp vật lý như tổng trở cao (Z) hay không xác định (X). Các hằng số cũng được hổ trợ như là SC_LOGIC_1, SC_LOGIC_0, SC_LOGIC_X và SC_LOGIC_Z. Để ít đánh máy hơn, nếu sử dụng "namespace" sc_dt, thì sau đó chỉ cần dùng Log_1 và Log_0, Log_X, Log_Z hoặc '1', '0', 'X', 'Z

Bởi vì những điều trên, những kiểu dữ liệu này chậm hơn đáng kể so với sc_bit và sc_bv. Để công việc tốt nhất nên luôn sử dụng kiểu giống như bool.

sc_logic **NAME[,NAME]…;**

sc_lv<BITWIDTH> **NAME[,NAME]…;**

SystemC không biểu diễn kiểu dữ liệu nhiều mức(multi-level) hoặc "drive strengths" giống như Verilog có 12 mức logic hoặc VHDL là 9 mức giá trị std_logic. Tuy nhiên, bạn có thể tạo tùy chọn dữ liệu nếu cần thiết, có thể vận dụng chúng bằng việc điều hành(operator) việc quá tải (overloading) trong SystemC.

Ví dụ:

sc_logic buf(sc_dt::Log_Z);

sc_lv<8> data_drive("zz01XZ1Z");

data_drive.range (5,4) = "ZZ";

buf ='1';

2.1.5. Dấu chấm tĩnh

SystemC kiểu dữ liệu dấu chấm tĩnh: sc_fixed, sc_ufixed, sc_fix, sc_ufix, và _fast biến thể của chúng.

2.1.6. Operator for SystemC data type

SystemC hỗ trợ tất cả kiểu dữ liệu chung hoạt đông với "operator overloading"

Table 4-4: Operator

Comparison- So sánh	==	!=	>	>=	<	<=	
Arithmetic- toán học	++	--	*	/	%	+	-

Bitwise	~	&	|	^						
Assignment	=	&=	|=	^=	*=	%=	+=	-=	<<=	>>=

Theo đó, SystemC cũng cung cấp các phương thức đặc biệt " access bit, bit range' và thực hiện chuyển đổi rõ ràng.

Table 4-5: Special methods

Bit Selection	Bit(idx), [idx]
Range Selection	Range(high,low), (high,low)
Conversion	To_double(), to_int(), to_int64(), to_long(), to_unit(), to_uint64(),to_ulong(), to_string(type)
Testing	Is_zero(), is_neg(), length()
Bit reduction	and_reduce(), or_reduce(), nand_reduce(), nor_reduce(), xor_reduce(), và xnor_reduce().

Ví dụ: Conversion Issues

```
sc_int<3>    d(3);
sc_int<5>    e(15);
sc_int<5>    f(14);
sc_int<7>    sum= d+e+f ;//work
sc_int<64>    g('0x7000000000000000");
sc_int<64>    h("0x7000000000000000");
sc_int<64>    i("0x7000000000000000");
sc_bigint<70>  bigsum= g+h+i; //doesn't work
bigsum = sc_bigint<70> (g)+h+i; //work
```

2.1.7. Mức trừu tượng cao hơn và STL

The **Standar Template Library (STL)** là thư viện phổ biến nhất, nó có thể "đi với" mọi trình biên dịch C++ mới nhất. STL có nhiều kiểu dữ liệu và cấu trúc hữu ích, bao gồm cải thiện mãng kí tự như string.

STL có chứa nhiều loại như: vector<>, map<>, list<> and deque<>, mà có thể có nhiều kiểu dữ liệu khác nhau. STL có thể sử dụng các hàm thuật toán có sẵn như là for_each(), count(), min_element(), max_element(), search(), transform(), reverse() và sort(). Đó là một trong số ít thuật toán có sẵn.

STL hổ trợ kiểu vector<> thì giống như mảng array của C++ với một vài sự cải thiện. Thứ nhất, vector<> có thể thay đổi kích thước. Hai, và có lẻ quan trong hơn, truy cập yếu tố vector<> có thể có giới hạn để đảm bảo an toàn.

Ví dụ:

```
#include <vector>
Int main(int argc, char* argv[]){
    std::vector<int> mem(1024);
    for(unsigned i=0; i!=1024; i++) {
    //
    Mem.at(i) = -1;     //
    }
….
Mem.resize(2048);      // tang kích thước bộ nhớ
…..
}
```

Ví dụ:

```
#include <iostream>
#include <map>
Int main(int argc, char* argv[]){
    typedef unsigned long ulong;
    std::map<ulong, int> lmem;
    while (lmem.size()< 10) {
```

```
        lmem[rand()]=rand();
    }
    typedef std::map<ulong, int>::const_iterator iter;
    for (iter iv=lmem.begin(); iv!=lmem.end(); iv++) {
        std::cout<< std::hex
            <<"lmem["<< iv -> first
            <<"]=" << iv->second << ";" << std::end1;
    }
}
```

2.1.8. Chọn đúng kiểu dữ liệu

Câu hỏi được đặt ra, "Kiểu dữ liệu nào cần được sử dụng cho việc thiết kế?.

Câu trả lời tốt nhất là: "Chọn kiểu mà gần kiểu "native C++"nhất có thể để mô phỏng".

Thứ tự ưu tiên được thể hiện.

Fastest: Native C/C++ Data Type (int, double and bool)

sc_int<>, sc_uint<>
sc_bit, sc_bv<>
sc_logic, sc_lv<>
sc_bigint<>, sc_biguint<>
sc_fixed_fast<>, sc_fix_fast, sc_unfixed_fast<>, sc_ufix_fast

Slowest: sc_fixed<>, sc_fix, sc_ufixed<>, sc_ufix.

2.2. Module và Class

2.2.1. Điểm bắt đầu sc_main và cấu trúc file

2.2.1.1. Điểm khởi đầu mô phỏng

Khi mô tả phần cứng bằng ngôn ngữ Verilog/VHDL, việc mô phỏng hành vi của phần cứng bắt đầu bởi việc viết file testbench để thay đổi các giá trị tín hiệu ngõ vào. Tín hiệu ngõ vào thay đổi sẽ lan truyền bên trong thiết kế và truyền đến ngõ ra tương tự tín hiệu điện truyền từ trái sang phải. Việc kết thúc mô phỏng theo thời gian là việc kết thúc của thủ tục "initial" hay có hàm $finish được gọi.

Bản chất mô phỏng theo thời gian thực nghĩa là tại một thời điểm các biến/dây/… được tính và update vào bộ nhớ. Thời điểm kế tiếp làm tương tự. Cuối cùng tập các giá trị tại các thời điểm khác nhau được vẽ lên tạo dạng sóng cho cảm giác tín hiệu chạy song song và update liên tục. Việc kiểm soát và lên lịch trình hay khoảng cách giữa các thời điểm (ns hay us hay …) được trình biên dịch kiểm soát và lên lịch trình. Bản chất tại một thời điểm (ns thứ 1, thứ 2 …) dây/biến nào được cập nhật thì người thiết kế không thể biết được, chỉ có trình biên dịch mới biết vì cơ bản trình biên dịch vẫn thực thi theo thứ tự từng tác vụ một. Do đó, với một biến và một dây trong thiết kế phân cứng chỉ nên được gán giá trị ở một thủ tục duy nhất (gán bới một assign hay always) để kiểm soát giá trị tại một thời điểm nhất định. Nếu biến được gán ở hai thủ tục khác nhau (ở hai always) thì trình biên dịch một sẽ báo lỗi (trình biên dịch tốt) hai là không báo lỗi nhưng cập nhật giá trị nào (vòng always nào được thực hiện trước) cho thủ tục nào thì người thiết kế không thể biết ✉ lỗi cho thiết kế.

Trong C++, việc 1 chương trình bắt đầu thực hiện sẽ là hàng đầu tiên của hàm **main**. Nếu dùng ngôn ngữ C++ và SC để mô tả một phần cứng thì cũng có thể hiểu hàm main như file testbench.

Trong hàm main sẽ gọi (instance) các lớp (class) mà trong đó sẽ có một lớp chính là thiết kế. So với ngôn ngữ Verilog, việc gọi một lớp trong hàm main của C++ giống như việc gọi một module trong file testbench của ngôn ngữ Verilog.

Hàm main của chương trình C được viết như sau

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

```
}
```

+ Theo trên, **argc** đại diện số dòng lệnh của chính chương trình đó.

+ Tham số thứ 2, argv[], một mảng của C chuẩn và là những chuỗi ký tự đang đại diện cho dòng lệnh mà ta gọi là chương trình.

+ argv[0] chứa tên của chương trình.

Dựa trên kiểu viết này, SC cung cấp một sự thay thế được biết như **sc_main().**

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

+ Thường thì hàm sc_main sẽ được viết trên file tên là main.cpp

+ Do SC dùng để mô tả phần cứng, mà phần cứng thì sẽ chạy liên tục khi được cấp điện với xung clock liên tục. Để có thể mô hình phần cứng, SC và trình biên dịch phải thiết lập một time schedule (biểu đồ thời gian) mà trên biểu đồ này có đơn vị nhất đinh. Ví dụ đơn vị đó là nano giây. Thì cứ sau mỗi nano giây, các biến được tính toán và cập nhật lại lên bộ nhớ. Bản chất khi người lập trình xem giá trị các biến trên màn hình tại một thời điểm nhất định (ví dụ các giây thứ 5, 10, 15 chẳng hạn) có cảm giác thấy các giá trị này update song song nhưng thật ra chúng được tính toán tuần tự nhưng khi xuất ra màn hình thì sẽ dựa trên thời điểm xuất để xuất ra cùng lúc tạo cảm giác biến update song song.

Người dùng SC phải quản lý tốt thời gian thiết lập, sử dụng nhuần nhuyễn các phương thức chạy song song tránh xung đột các biến dữ liệu và quản lý thật tốt bộ nhớ để trình biên dịch có thể thực hiện trơn tru trên biểu đồ thời gian này.

Hàm sc_start() cho phép chương trình bắt đầu chạy nghĩa là khi hàm này được gọi giống như bắt đầu cấp điện cho thiết kế và nguồn clock bắt đầu có. Việc ngõ vào input của thiết kế thay đổi như thế nào để mô phỏng thiết kế phụ thuộc vào clock bắt đầu tức có nghĩa là khi sc_start() được gọi. Cụ thể khi viết các trường hợp kiểm tra thiết kế, người viết cũng sẽ chỉ đẩy các tín hiệu kiểm tra ngõ vào khi có xung clock để có thể đồng bộ với thiết kế mà xung clock chỉ được bắt đầu khi hàm sc_start() bắt đầu. Như vậy dùng hàm sc_start() để bắt đầu cho tính đồng bộ cao và giống như phần cứng bắt đầu được cấp điện. Có thể hiểu khi hàm

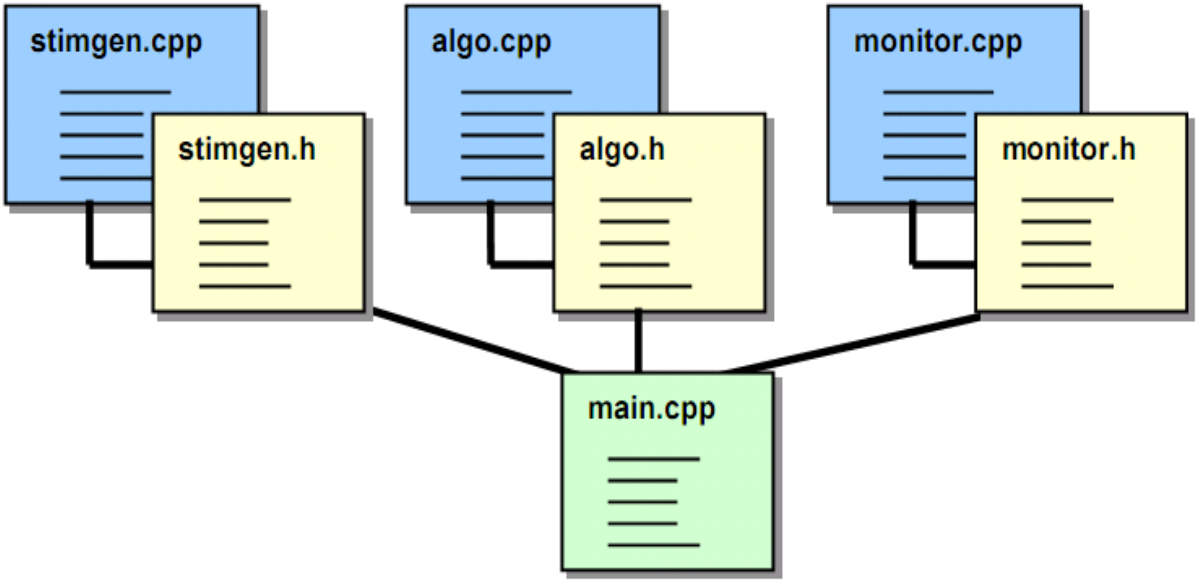
này được gọi và bắt đầu mô phỏng giống thời điểm đầu tiên của thủ tục "**initial**" trong **Verilog.**

Trước khi sc_start() được gọi là phần <ELABORATION>, tại phần này thiết kế được gọi (instance) và thiết kế dùng để kiểm tra cũng được gọi. Việc gọi các thiết kế ra trong hàm main nghĩa là gọi lớp (Class).

```
int sc_main(intargc,char* argv[]) {
    and_gate_logic and_gate_logic_01 ("I am logic_and");
    sc_start(); // <-- Simulation begins & ends in this function!
    return 0;
}
```

Trong ví dụ trên lớp "and_gate_logic" được gọi ra với tên định danh là "and_gate_logic_01". Mỗi lớp đều có hàm dựng của riêng của nó và hàm dựng là một hàm bên trong của lớp và bắt buộc phải có. Hàm dựng có tên trùng với tên lớp. Khi lớp được gọi ra ở bất cứ đâu thì hàm dựng sẽ sẽ được gọi ngay tức khắc. Vì hàm dựng trong trường hợp này nhận đối số là chuỗi nên lớp "and_gate_logic" phải truyền một chuỗi là "I am logic_and" cho hàm dựng. Chi tiết sẽ được mô tả sau khi đi sâu về cấu trúc một lớp.

Cấu trúc các file lập trình được mô tả bởi hình vẽ dưới đây với sc_main làm gốc sẽ gọi các file header (file.h) và file thực thi (file.cpp)



Trong các file.cpp sẽ gọi tiếp các lớp khác, tầng tầng lớp lớp việc gọi các lớp được thực hiện làm chương trình phình to, người kỹ sư phải hiểu rõ cấu trúc cây thư mục của toàn bộ thiết kế trước khi thực hiện các tác vụ khác.

2.2.1.2. Cấu trúc, khai báo và cách gọi một lớp

Để có thể hiểu cấu trúc cơ bản của một lớp, ví dụ sau được đưa ra:

```
#include <iostream.h>
class cat
{
public:
   int itsAge;
   int itsWeight;
};
int sc_main(intargc,char* argv[]) {
   cat cat_01;
    cat_01.itsAge = 5;
    cat_01.itsWeight = 20;
}
```

Phân tích: Lớp "cat" không có hàm, chỉ có 2 biến "itsAge" và "itsWeight". Thực tế là khi không khai báo hàm dựng và hàm hủy thì trình biên dịch sẽ tự thêm vào. Khi tự thêm vào thì hàm dựng sẽ không có đối số đưa vào. Hàm hủy không nhận đối số. Ví dụ sau cho thấy người dùng có khai báo hàm dựng và hủy.

```
#include <iostream.h>
class cat
{
public:
   int itsAge;
   int itsWeight;
  cat(); //hàm dựng
  ~cat(); // hàm hủy
};
cat::cat(){};
cat::~cat(){};

int sc_main(intargc,char* argv[]) {
   cat cat_01;
    cat_01.itsAge = 5;
    cat_01.itsWeight = 20;
```

```
}
```

Phân tích: Trong ví dụ trên, tuy hàm dựng và hàm hủy được người dùng viết ra nhưng hàm dựng không nhận đối số gì nên ở hàm sc_main, lớp chỉ đơn thuần được **gọi** "cat cat_01". Vì mặc định khi lớp được gọi ở bất cứ đâu (trong ví dụ trên gọi ở sc_main) thì trình biên dịch sẽ gọi hàm dựng. Khi hàm dựng được khai báo thì đoạn code thực thi hàm dựng phải được viết theo. Trong ví dụ này thì việc thực thi hàm dựng và hàm hũy cũng không có làm tác vụ gì.

Chính xác thì hàm dựng sẽ nhận đối số hoặc không nhận và khởi tạo hoặc không khởi tạo các biến trong lớp tùy theo mục đích sử dụng của lớp đó. Hàm hủy sẽ giải phóng bộ nhớ đặc biệt là các khai báo con trỏ hay các lớp con được gọi bên trong.

Ví dụ hàm dựng có thể sẽ thiết lập các giá trị ban đầu cho các biến của lớp (các biến itsAge, itsWeight) hay in ra một chuỗi nào đó. **Hàm hủy dùng để hũy lớp khi kết thúc chương trình và giải phóng bộ nhớ. Vậy câu hỏi đặt ra là bộ nhớ được sử dụng khi nào mà giải phỏng (Sẽ trả lời sau).**

Ví dụ sau đây cho thấy trong hàm dựng có hành vi:

```
#include <iostream.h>
class cat
{
public:
   int itsAge;
   int itsWeight;
  cat(int initialAge); //hàm dựng
  ~cat(); // hàm hủy
};
cat::cat(int initialAge){
 itsAge = initialAge;
};
cat::~cat(){};

int sc_main(intargc,char* argv[]) {
   cat cat_01(3);
```

```
    cat_01.itsAge = 5;
    cat_01.itsWeight = 20;
}
```

Phân tích: Trong ví dụ trên, ở hàm sc_main truyền tham số là 3 vào khi gọi lớp cat (cat cat_01(3)). Khi lớp này được gọi, hàm dựng cat() sẽ được thực thi. Khi hàm dựng được thực thi, nhìn vào cách thức mà hàm dựng được thực thi thì hàm này sẽ phải nhận một đối số "initialAge". Và đối số này sẽ gán cho biến itsAge. Vậy có thể thấy rằng đối số hàm dựng sẽ được truyền từ sc_main hay bất cứ chổ nào gọi nó

Quay lại câu hỏi khi bộ nhớ được sử dụng và sử dụng ra sao? Có thể hiểu rằng chương trình đơn thuần là một đoạn văn bản. Khi đoạn văn bản được biên dịch để tạo ra file binary (trong window là file exe) thì đơn thuần nó vẫn là đoạn văn bản chứa chuỗi dữ liệu binary 0 và 1. Chỉ khi file binary này được thực thi thì khi đoạn chương trình chạy trong sc_main tới lúc gọi lớp thì bộ nhớ vật lý sẽ cấp phát vùng nhớ để lưu trữ các biến có trong trong lớp này. Các hàm không được cấp phát bộ nhớ vì đơn thuần hàm là hành vi và hành vi thì sẽ được biên dịch thành các chuỗi binary hay còn gọi là mã lệnh lưu trữ trên vùng nhớ lệnh. Chỉ các biến được khởi tạo sẽ có vùng nhớ riêng cho nó. Vùng nhớ này khác với vùng nhớ lệnh. Để hiểu rõ điều này cần tham khảo kiến trúc của một CPU và quy trình mà một CPU thực thi các lệnh của nó.

2.2.2. Đơn vị cơ bản của thiết kế SC_MODULE

Nếu hiểu theo phần cứng thì SC_MODULE giống như khai báo một module trong ngôn ngữ Verilog, có ngõ vào và ngõ ra. Nội dung trong SC_MODULE chính là hành vi của phần cứng. Trong SC_MODULE sẽ có cách thức viết (các hàm, thủ tục) mô tả giống hành vi của cổng logic và DFF) để mô hình hóa hành vi chính xác của phần cứng.

Hiểu theo hướng ý nghĩa lập trình thì SC_MODULE chính là một lớp trong systemC, lớp này sẽ có các biến và hàm bên trong nó giúp nó có thể mô phỏng được hoạt động giống một phần cứng. Phân tích cách khai báo và nội dung của một SC_MODULE (kiểu mới) và một Class cổ điển (kiểu củ) sẽ cho thấy sự tương đồng.

2.2.2.1. Cách khai báo một SC_MODULE theo kiểu mới

```
#include <systemc.h>

SC_MODULE(module_name) {
```

MODULE_BODY

};

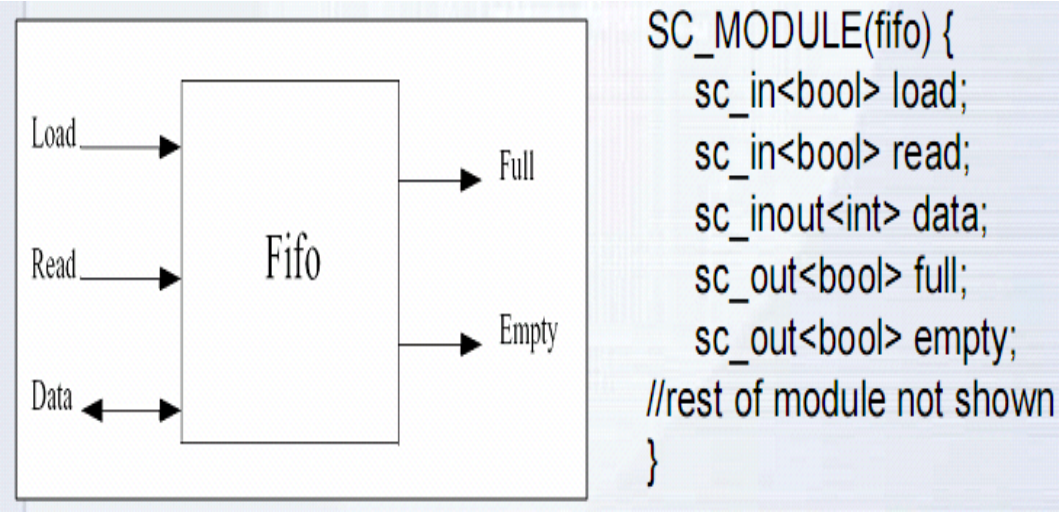
Hoặc sử dụng cú pháp “#define”

#define SC_MODULE(module_name)

struct module_name: public sc_module

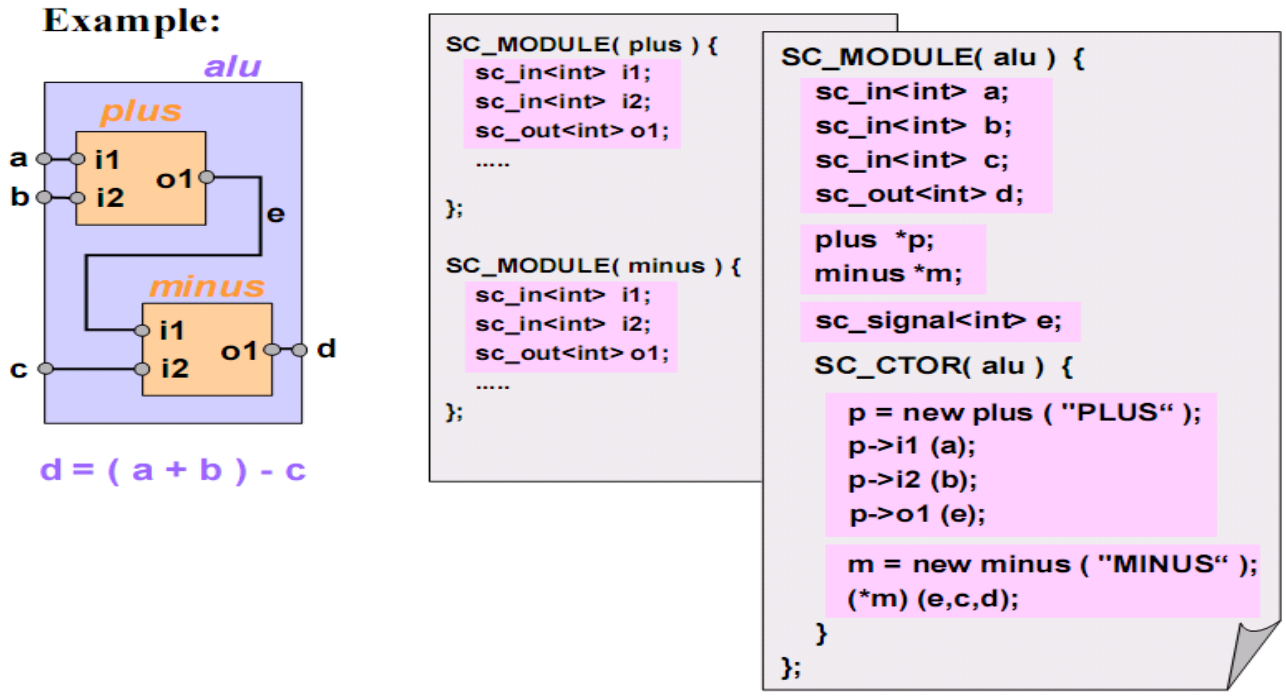
Bên trong lớp module này bao gồm:

- **Ports**
- **Member channel instances**: trường kênh
- **Member data instances**: trường dữ liệu – Các biến
- **Member module instances** (sub-designs – gọi một lớp/moudule khác)
- **Constructor**: hàm dựng (vì bản chất SC_MODULE là một lớp)
- **Destructor**: hàm hủy
- **Process member functions** (processes) – Trong System C ngoài biến và hàm có thêm khái niệm Process sẽ bàn sau
- **Helper functions**



Ví dụ hình vẽ trên cho thấy một khối phần cứng có tên là “Fifo” thì khai báo SC_MODULE tương ứng như bên phải với các port input và output là kiểu SC_IN/SC_OUT/SC_INOUT. Khi có hai thiết kế nối với nhau thì hình vẽ sau được mô tả.

Đối số truyền vào là “fifo” là chuỗi string. Khai báo dùng lớp tên SC_MODULE luôn yêu cầu tên định danh. Khai báo kiểu lớp “class” thì không bắt buộc



Trong SC_MODULE(alu), con trỏ p kiểu lớp “plus” và con trỏ m kiểu “minus” được khai báo. Khi khai báo kiểu con trỏ. Trong SC_CTOR (SC_CTOR chính là hàm dựng của lớp alu), con trỏ p và m mới chính thức được thiết lập (nghĩa là các biến trong lớp plus và minus chính thức được cấp phát bộ nhớ. Việc khai báo con trỏ ban đầu vẫn chỉ là đăng ký chứ chưa có ý nghĩa cấp phát vật lý. Trong hàm dựng SC_CTOR, việc nối dây được thực hiện. Dấu mũi tên là cách con trỏ sẽ truy xuất các thành phần của lớp tương ứng.

Chú ý, toàn bộ biến trong lớp minus và plus chỉ được cấp phát bộ nhớ khi được khởi tạo nghĩa là hàng lệnh có chữ “new” chứ không phải khai báo con trỏ kiểu plus hay minus ở hàng lệnh phía trên.

2.2.2.2. Cách khai báo một SC_MODULE theo kiểu cũ

Kiểu cũ là kiểu mà C++ vẫn thường dùng để khai báo một class như mẫu sau:

*//============================ **LỚP PLUS** ========================//*

***class** plus : **public** sc_module //Kế thừa sc_moudule ✉ Đoạn lệnh này không được gọi khi khai báo theo kiểu mới*

{

sc_in <int> i1;

sc_in <int> i2;

```
  sc_out <int> o1;

 char name;
   …….
  plus (sc_module_name name): sc_module (name) ;  //hàm dựng phải truyền biến tên "name" để đặt tên và khởi tạo sc_module
   ~plus(); // hàm hủy
};
plus::plus(){ //Hàm dựng
};
plus::~plus(){ //Hàm hủy
};
//============================LỚP MINUS=========================//
class minus: public sc_module
{
    sc_in <int> i1;
   sc_in <int> i2;
  sc_out <int> o1;
  char name;
   …….
  minus (sc_module_name name): sc_module (name); //hàm dựng
  ~ minus (); // hàm hủy
};

//==== Hàm dựng
minus:: minus (){
};
minus::~ minus (){
};

//============================LỚP ALU ========================//
class alu: public sc_module
{
    sc_in <int> a;
```

```
  sc_in <int> b;
 sc_in <int> c;

 sc_out <int> d;
 char name;

//===== Khai báo con trỏ===//
plus *p;
minus *m;

//====== Chân nối internal ====//
sc_signal<int> e;
  …….
 alu (sc_module_name name): sc_module (name); //hàm dựng
 ~ alu (); // hàm hủy
};
//================ thực thi hàm dựng ALU=========================//
alu:: alu (){

p = new plus("PLUS"); // Lúc này mới khởi tạo lớp plus
//===== Gán bằng dấu mũi tên theo tên ===//
 p->i1(a);
p->i2(b);
p->o1(e);
……..…
};
//======== Thực thi hàm hủy ALU ================================//
alu::~ alu (){
};
```

2.2.3. Hàm dựng của lớp SC_MODULE: SC_CTOR

Hàm SC_MODULE hay còn gọi là hàm dựng cho lớp SC_MODULE thực hiện nhiệm vụ đặc biệt của SystemC bao gồm:

- Khởi tạo/ cấp phát sub-design (Chapter 10, Structure)

- Nối những thiết kế nhỏ (Chapters 10, Structure and 11,Connectivity)
- Registering processes with the SystemC kernel (Chapter 7,Concurrency)
- Providing static sensitivity (Chapter 7, Concurrency)
- Miscellaneous user-defined setup

Cấu trúc hàm SC_CTOR:

```
SC_CTOR(module_name)
:Initialization //OPTIONAL
{
Subdesign_Allocation
Subdesign_Connectivity
Process_Registration
Miscellaneous_Setup
}
```

Nếu khai báo theo kiểu củ thì không cần từ khó SC_CTOR.

2.2.4. Đơn vị chấp hành cơ bản: SystemC process

Để có thể mô phỏng hành vi của phần cứng, SC hổ trợ các process bao gồm 2 thủ tục là SC_METHOD và SC_THREAD sẽ được nói chi tiết sau. SC_METHOD và SC_THREAD cũng có các khai báo đối số đầu vào giống như đối tượng kịch thích nhằm thực hiện giống như các ngõ vào cổng logic hay clock, reset để kích thích và bắt đầu thực hiện một hành vi nhất định như các logic/DFF.

Một ví dụ sau cho thấy chi tiết:

```
class sc_and : public sc_module        //Ke thua sc_module kieu public, neu khong co public thi la private
{
public:
    sc_in<sc_lv<N_INPUTS>> a;    //Template N_INPUT kieu int
    sc_in<sc_lv<N_INPUTS>> b;
    sc_out<sc_lv<N_INPUTS>> c;
```

```
SC_HAS_PROCESS(sc_and);  // Viet theo kieu cu phai dang ky line nay cho biet trong ham dung co dunug process SC_METHOD
sc_and (sc_module_name name): sc_module (name)  // Truyen name kieu sc_module_nam vao ham dung, tu ham dung truyen name vao ham dung cua sc_module ; Hoac
//SC_CTOR (sc_and)
{
        SC_METHOD(do_and);
        sensitive << a << b;
}
//~sc_and();
void do_and()
{
        c.write(a.read() & b.read());
}
};
```

Hành vi and logic cho hai tín hiệu a và b được thực hiện trong hàm "void do_and()" nhưng hàm này được đăng ký bởi đoạn lệnh "SC_METHOD(do_and)" có nghĩa là process có giao thức SC_METHOD sẽ gọi hành vi trong hàm "do_and". Process này nhận hai đối số là a và b với dòng lệnh "sensitive << a << b;" nghĩa là khi có a hay b thay đổi. Để có thể dùng được thủ tục SC_METHOD thì SC_HAS_PROCESS(sc_and) phải được khai báo.

Có hai thủ tục chính là SC_METHOD và SC_THREAD sẽ được bàn chi tiết dưới đây.

2.2.5. Ghi tiến trình đơn giản: SC_THREAD

SC_THREAD(process_name); *// Khai báo SC_THREAD này bắt buộc nằm trong hàm dựng SC_CTOR như ví dụ sau*

Ví dụ:

```
//FILE: simple_process_ex.h

SC_MODULE(simple_process_ex) {

        SC_CTOR(simple_process_ex) {

        SC_THREAD(my_thread_process);

        }

void my_thread_process(void);

};
```

Theo đó, tên của file .h giống với tên khai báo trong SC_THREAD

```
//FILE: simple_process_ex.cpp

Void simple_process_ex::my_thread_process(void) {

        std::cout<< "my_thread_process executed within "

                <<name()

                <<std::endl;

}
```

2.2.6. Hoàn thành thiết kế đơn giản main.cpp

Ví dụ: *simple_process_ex*

```
//FILE: main.cpp

int sc_main(intargc,char* argv[]) { // args unused

        simple_process_ex my_instance("my_instance"); // Khởi tạo lớp simple_process_ex

        sc_start(); //Bắt đầu mô phỏng

        return0;  // unconditional success (not recommended)

}
```

2.2.7. Hàn dựng thay thế SC_HAS_PROCESS

Sử dụng SC_HAS_PROCESS thay thế SC_CTOR khi bạn có yêu cầu tạo tham số trong trường hợp tên của chuỗi vượt quá so với SC_CTOR và muốn thay thế hàm dựng (constructor) thành hàm triển khai. Bạn có thể sử dụng hàm tạo tham số để xác định kích thước bao gồm: memories, address ranges for decoder, FIFOs, chia clock, FFT và cấu hình thông tin khác.

```
//FILE:module_name.h

SC_MODULE(module_name) {

        SC_HAS_PROCESS(module_name);

        module_name(sc_module_nameinstname[,other_args…]):sc_module(instname)
```

[,other_initializers]

{

CONSTRUCTOR_BODY {

};

//FILE:module_name.h

SC_MODULE(module_name) {

SC_HAS_PROCESS(module_name);

module_name(sc_module_nameinstname[,other_args…]);

};

//FILE:module_name.cpp

module_name::**module_name**(

sc_module_name instname[,other_args…])

:**sc_module**(instname)

[,other_initializers]

{

CONSTRUCTOR_BODY

}

2.3. Khái niệm thời gian trong SystemC

2.3.1. sc_time

SystemC cung cấp kiểu dữ liệu sc_time để đo thời gian. Thời gian được biểu diễn 2 phần: phần giá trị và đơn vị thời gian.

Các đơn vị thời gian: **SC_SEC, SC_MS, SC_US, SC_NS, SC_PS, SC_FS.**

Cấu trúc khai báo:

sc_time **name;** // không khởi tạo giá trị

sc_time **name(giá tri, đơn vị)** // có giá trị khởi tạo

SystemC cho phép cộng, trừ, nhân và những hoạt dộng có liên quan đến **sc_time**.

Ví dụ:

sc_time t_period(5, **sc_ns**);

sc_time t_timeout(100, **sc_ms**);

sc_time t_measure, t_current, t_last_clock;

t_measure= {t_current – t_last_clock };

if (t_measure> t_hold) {error("setup violated")}

Lưu ý các quy ước thường được sử dụng cho các biến thời gian. Quy ước này hỗ trợ cho việc hiểu được code. Bên cạnh đó, việc sử dụng những hằng số là không khuyến khich vì nó làm giảm việc tường minh và ý nghĩa.

Một hằng số đặc biệt cần được chú ý là **SC_ZERO_TIME**, nó là kiểu đơn giản của **sc_time**(0, **SC_SEC**).

2.3.2. sc_start()

sc_start là phương thức quan trọng trong systemC. Phương thức này khởi đầu cho bước mô phỏng, nó bao gồm việc khởi tạo và thực hiện. Điểm thú vị của chương này, sc_start() có một tùy chon tham số của sc_time. Với việc không có tham số, sc_start() quy định việc mô phỏng là mãi mãi. Nếu bạn cung cấp tham số thời gian, mô phỏng sẽ ngừng lại sau thời gian quy định.

sc_start(); // sim forever

sc_start(max_sc_time) // cài đặt thời gian mô phỏng tối đa.

Ví dụ:

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

{ simple_process_ex my_instance("my_instance");

***sc_start**(60.0, **SC_SEC**); // Chạy trong vòng 60 giây*

***return** 0;}*

Lưu ý rằng trong nội bộ, SystemC biểu diễn thời gian với 64bit integer(**int64**). Kiểu dữ liệu này có thể biểu diễn thời gian rất dài nhưng không vô hạn.

2.3.3. sc_time_stamp() và hiển thị thời gian (time display)

SC dùng hàm sc_time_stamp() để xuất thời gian hiện tại đang biên dịch cho phép in ra giá trị thời gian này.

Cout<< sc_time_stamp()<<end1;

Ví dụ:

***Std::cout**<< "the time is now "*

*<< **sc_time_stamp()***

*<< "!" << **std::end1**;*

2.3.4. wait(sc_time)

wait() thường dung để làm trễ(delay) tiến trình đi một khoảng thời gian. Có thể sử dụng sự trễ này để mô phỏng một cách chậm rãi các hoạt động thực tế (hoạt động máy móc, thời gian phản ứng hóa học, tín hiệu lan truyền…). Phương thức **wait()** cung cấp cú pháp cho phép trì hoãn cụ thể một khoảng thời gian trong tiến trình **SC_THREAD (SC_THREAD** process). Khi wait()được gọi, **SC_THREAD** process ngưng lại rồi sau đó chạy tiếp tục sau khoảng thời gian delay. Chúng ta bàn luận về **SC_THREAD** process chi tiết hơn trong chương sau.

Wait(delay_sc_time);

Ví dụ:

void simple_process_ex: my_thread_process(**void**) {

wait(10, **sc_ns**);

std:: cout<< "Now at" << **sc_time_stamp()** << **end1**;

sc_time t_delay (2, **sc_ms**);

```
t_delay *=2     // t_delay=t_delay*2
std::cout<< “delaying”<< t_delay<< std::end1;
wait(t_delay);
std::cout<< “Now at” << sc_time_stamp()
        << std::end1;
```

// run example

Now at 10 ns

Delaying 4ms

Now at 4000010 ns

2.3.5. sc_simulation_time() và time resolution và time_unit:

Để thiết lập đơn vị thời gian mặc định, gọi hàm sc_set_default_time_unit() và phải được gọi trước tất cả đặc tả thời gian và sc_start(). Đứng trước hàm này là quy định độ phân giải (resolution) thời gian bằng việc sử dụng sc_set_time_resolution().

sc_set_time_resolution(value, tunit);

sc_set_default_time_unit(value, tunit);

Làm tròn sẽ xảy ra nếu bạn quy định hằng số thời gian trong phạm vi code có resolution nhiều hơn **resolution** quy định bởi thông thường. Ví dụ, nếu quy định time resolution là 100ps, coding là 20ps thì giá trị làm tròn là 0.

Ví dụ:

```
int sc_main(int argc, char* argv[]){
    sc_set_time_resolution(1,sc_ms);
    sc_set_default_time_unit(1,sc_sec);
    simple_process_ex my_instance(“my_instance”);
    sc_start(7200,sc_sec);
```

```
double t = sc_simulation_time();
unsigned hours = int(t/3600.0);
t -= 3600*hours;
unsigned minutes = int(t/60.0);
t -= 60*minutes;
double seconds = t;
cout    << hours<<"hours"
        <<minutes<<"minutes"
        <<seconds<<"seconds"
        end1;
return 0;
```

2.4. Process

2.4.1. Giới thiệu

Có hai process được sử dụng trong SC là SC_METHOD và SC_THREAD. Bảng so sánh sau đây cho thấy sự giống nhau và khác nhau giữa hai processes.

Mục đích SC hỗ trợ hai processes này nhằm giúp người dùng có thể mô tả các hành vi thực tế ngoài tự nhiên, các trạng thái, các hành động lặp đi lặp lại theo một quy trình nhất định

Đặc tính	SC_THREAD	SC_METHOD
Cách thức được kích thích/gọi để chạy/thực thi	Được kích thích bởi các cách thức sau: + Dùng "sensitive" list để liệt kê các biến mà nếu các biến này thay đổi sẽ kích thích process chạy + Dùng hàm "**notify**" để gọi	Giống SC_THREAD

	bất cứ lúc nào	
Kiểu chạy/thực thi	Nếu được gọi/kích thích sẽ chạy một lần rồi kết thúc trong một lần chạy mô phỏng toàn chương trình cho dù có các kích thích nhiều lần sau	Nếu được gọi/kích thích sẽ chạy một lần rồi kết thúc, nhưng sẽ được gọi và chạy lại nhiều lần khác khi các biến/hàm kích thích tác động những lần sau
So sánh với ngôn ngữ Verilog	Giống thủ tục Initial	Giống thủ tục always

Do SC_THREAD chỉ chạy một lần rồi thôi, người lập trình có thể sử dụng vòng lặp do…while để gọi hành vi SC_THREAD lại lần nữa mà không thoát ra. Sau đây là các ví dụ về SC_THREAD và SC_METHOD.

Ví dụ SC_METHOD:

```
SC_MODULE (test){
sc_event request_01;   // event 1
sc_event request_02;  // event 2
SC_CTOR (test){
   SC_METHOD (test_method)
   sensitive << request_01 <<request_02;
       dont_initialize();
}
void test_method (){
}
```

Phân tích: SC_METHOD nhận kích thích là biến request_01 và request_02 được khai báo kiểu sc_event. Khi các biến này kích thích, hành vi của process SC_METHOD sẽ thực thi và hành vi đó chính là hành vi của hàm test_method. Khi hàm test_method hoàn thành, SC_METHOD lại tiếp tục đợi kích thích cho lần chạy sau đó.

Người lập trình phải chú ý rằng nếu hàm test_method đang thực thi mà một kích thích mới tác động thì câu hỏi đặt ra là kích thích đó sẽ bị bỏ đi không quan tâm hay SC_METHOD chạy xong sẽ quay lại chạy một lần nữa ---> Chính xác điều này phụ thuộc vào trình biên dịch, tuy nhiên việc bỏ đi kích thích khi SC_METHOD đang thực thi được sử dụng nhưng người dùng tránh tạo ra việc này gây ra lỗi tiềm ẩn không mong muốn.

Kiểu kích thích này được gọi là static.

Ví dụ SC_THREAD:

```
SC_MODULE (test){
sc_event request_01;   // event 1
sc_event request_02;  // event 2
SC_CTOR (test){
   SC_THREAD (test_method)
   sensitive << request_01 <<request_02;
       dont_initialize();
}
void test_method ();
}

void test::test_method (){
   for (; ;){
      do {
         ....
         Wait();
      }while(<condition>);
   }
```

}

Phân tích: Do SC_THREAD chỉ chạy một lần, vòng for được sử dụng để hàm test_method chạy liên tục. Điều này nghĩa là SC_THREAD được gọi một lần và chạy miết không thoát ra. Hàm wait() trong vòng do-while đợi cho đến khi tín hiệu kích thích request_01 hay request_02 được kích thích lần hai ✉ Đây là kiểu static.

Vấn đề xung đột về việc kích thích nhiều lần trong quá trình process đang thực hiện ở SC_THREAD là tương tự với SC_METHOD. Người lập trình nên tránh trường hợp này trong quá trình lập trình.

Có một vấn đề nữa cho cả hai process là tại thời điểm mô phỏng ban đầu, làm sao biết là tại thời điểm đó các kích thích thay đổi hay không. Không ai mong muốn các process được gọi ngay từ thời điểm ban đầu. Để ngăn cản việc này, hàm **dont_initialize()** được gọi để báo trình biên dịch xử lý vấn đề này và được khai báo ngay bên dưới sensitive list.

2.4.2. sc_event

SC sử dụng lớp sc_event để tạo ra các đối tượng nhằm kích thích các process chạy ở một thời điểm nhất định. Nói cách khác các processes đợi các event kích thích để được thực thi.

Cú pháp: sc_event event_name [, event_name]…;

Có thể có nhiều event để kích thích một process.

2.4.3. Cách thức trình biên dịch thực thi

Để có thể hiểu trình biên dịch thực thi như thế nào, hình dưới mô tả chi tiết. Cụ thể sau khi hàm sc_start() được gọi, một trạng thái initial được nêu ra cho việc thiết lập các hàm dựng mà trong đó các biến/hàm được đăng ký và lưu trữ.

Sau trạng thái initial thì các process và các hàm được thực hiện. Các hàm trong C++ sẽ được gọi tuần tự khi gọi ra nhưng các process được xử lý song song. Về bản chất tất cả mọi thứ đều chạy tuần tự. Vậy làm cách nào để cho người lập trình hình dung quá trình song song của các process.

Các process sẽ được gọi bởi nhiều phương thức khác nhau như tự động, bị kích thích… và có thể nhiều process cùng chạy cùng lúc. Trong mỗi process có các biến giá trị được thay đổi. Vậy làm sao người lập trình hình dung ra được các biến này thay đổi lúc nào? Cùng một thời điểm 2 biến trên 2 processes khác nhau thì biến nào được cập nhật trước?

Chính xác trình biên dịch sẽ có cơ chế cập nhật các biến tại cùng một thời điểm và xuất ra màn hình dưới dạng report hay xung nhịp làm người dùng cảm giác các process đang chạy song song. Tuy nhiên thực tế các biến được cập nhật tuần tự. Điều này dẫn đến việc cần quan tâm đến đơn vị thời gian mà trình biên dịch sử dụng.

Ví dụ đối với trình biên dịch, đơn vị 1ns là nhỏ nhất. Điều này có nghĩa là cứ 1 ns thì trình biên dịch sẽ cập nhật các biến giá trị 1 lần và lưu trữ lại tại điểm thời gian đó. Sau khi chay mô phỏng xong giả dụ là 10 ns thì có tất cả 10 giá trị được lưu trữ cho một biến trên từng ns. Điều này là tương tự cho các biến ở các process khác. Sau đó, tất cả các biến này được vẽ lại bằng giảng đồ xung giúp người lập trình cảm giác tại một thời điểm, các biến thực hiện cùng lúc.

2.4.4. Dynamic sensitive và static sensitive

Có thể chia làm hai cách gọi một process là gọi kiểu dynamic và kiểu static. Kiểu static đã được thể hiện ở ví dụ ở trên.

2.4.4.1. Static sensitive

Cụ thể kiểu static được khai báo với từ khóa sensitive và theo sau đó là các event được liệt kê. Có một hay nhiều event là điều bình thường.

So sánh với Verilog cho việc thiết kế cổng logic and 2 ngõ vào (a và b). Khi a hoặc b thay đổi bất kỳ lúc nào ngõ ra (c) sẽ được cập nhật. Nếu dùng a và b như là kiểu dữ liệu sc_event được liệt kê trong danh sách sensitive đối với SC_METHOD và SC_THREAD là tương tự về mặt hành vi.

Sensitive list cho phép các kích thích dạng cạnh lên và cạnh xuống giống như xung clock hay reset để mô tả các khối tuần tự/DFF…

2.4.4.2. Dynamic sensitive

Để có thể hiểu được kiểu gọi này, xét ví dụ các process gọi nhau thực thi. Cụ thể có 2 process A và B, A sẽ gọi B thông qua một event A_trigger_B và B sẽ gọi A thông qua event B_trigger_A. Hình thức này giống như máy trạng thái chuyển từ A qua B và ngược lại. Có thể hiểu việc kích thích nhau qua lại giữa các process gần giống với các trạng thái gọi nhau trong Verilog. Hành vi bên trong của Process chính là hàm thực hiện thay đổi các giá trị của biến tương đương với các khối combination logic để thay đổi ngõ ra tại một trạng thái nhất định.

Ví dụ:

```
SC_MODULE (test){
sc_event request_01;   // event 1
sc_event request_02;  // event 2
SC_CTOR (test){
   SC_THREAD (test_thread)
   sensitive << request_thread;
   dont_initialize();

   SC_METHOD (test_method)
   sensitive << request_method;
   dont_initialize();

}
void test_thread ();
}
void test_method ();
}
void test::test_thread (){
   for (; ;){
      do {
         ....
         request_method.notify();
         Wait(request_thread);
      }while(<condition>);
   }
}
```

```
void test::test_method (){
    ....
    next_trigger(4);
    request_thread.notify();
}
```

Phân tích: SC_THREAD được kích thích bởi request_thread được gọi trong SC_METHOD và SC_METHOD được kích thích bởi request_method được gọi trong SC_THREAD. Cách gọi là dùng hàm notify(). Đây là phương thức gọi kiểu dynamic.

Trong SC_THREAD có nhiều cách gọi khác nhau được mô tả và có thể kích thích ngay tại thời điểm ban đầu hay sau một khoảng thời gian

```
request_thread.notify();
request_thread.notify(SC_ZERO_TIME);
request_thread.notify(time);

notify(event_name);
notify(event_name, SC_ZERO_TIME)
notify(event_name, time)
```

Kết hợp với hàm wait trong SC_THREAD ta có

```
wait(time);
wait(event);
wait(event_01 & event_02); // Phai ca 2 event
wait(event_01 | event_02); // Hoac 1 trong 2
wait(timeout, event);  // Co event nhung phai doi sau do mot khoang thoi gian timeout
wait(timeout, event_01 | event_02);
```

Vậy trong SC_THREAD, hàm wait nếu có các đối số event thì sẽ là dynamic, các event này được gọi (notify) ở các process khác. Nếu không có các event trong wait mà đợi từ sensitive list thì được gọi là static

Phân tích: SC_METHOD được kích thích bởi request_method được gọi trong SC_THREAD. Hàm next_trigger tương tự wait trong SC_THREAD

next_trigger(time);

next_trigger(event);

next_trigger(event_01 & event_02); // Phai ca 2 event

next_trigger(event_01 | event_02); // Hoac 1 trong 2

next_trigger(timeout, event); // Co event nhung phai doi sau do mot khoang thoi gian timeout

next_trigger(timeout, event_01 | event_02);

2.5. Kiểu dữ liệu kênh truyền (Basic channel)

2.5.1. Primitive channels

Hiểu theo khía cạnh ngôn ngữ thì **channel** là một lớp đơn thuần có kế thừa lớp cơ bản **"sc_prim_channle".**

Hiểu theo khía cạnh phần cứng thì channel được thiết lập để nối các **module**/phần cứng ✉ Ở module top thì **channel** như là các dây nối trung gian. Có thể hiểu các **module** con có giao diện là kiểu **port** (port in/ port out) và các **port** này sẽ nối với các port khác của module khác. Trong Verilog, các dây trung gian luôn được khai báo **wire** trong module top khi thiết kế cấp độ **RTL**. Trong SC thì các dây trung gian này sẽ là kiểu **channel**.

SystemC's primitive channel không có chứa phân cấp, không có bất cứ "process" nào được gọi bên trong lớp này. Một số kiểu **channel** được giới thiệu

2.5.2. sc_mutex

sc_ mutex NAME; //Kiểu này hổ trợ các hàm **clock**, **tryclock** và **unclock**

// Khóa mutex NAME chờ đến khi mở nếu sử dụng

NAME.**lock**();

//Không chặn, trả về true nếu thành công, ngược lại là false

NAME.**trylock**();

// Mở khóa

NAME.**unlock**();

Ví dụ:

sc_mutex drivers_seat;

…

car->drivers_seat.**lock**(); // khóa nếu có người mua

car-> **start**();

….// điều hành xe

car-> **stop**();

car->drivers_seat.**unlock**(); // mở nếu người nào đó rời khỏi xe

Trong thiết kế điện tử ứng dụng, có thể áp dụng **sc_mutex** cho hành vi điều phối xe bus mà phải cần nhiều người điều khiển mới làm được việc này. Thay cho thiết kế người điều phối (arbiter), sc_mutex có thể sử dụng để quản lý tài nguyên xe bus nhanh nhất cho tới khi arbiter sắp xếp hoặc thiết kế.

Ví dụ: sử dụng class

```
class bus{
   sc_mutex bus_access;
   …
   void write(int addr, int data) {
        bus_access.lock();
        bus_access.unlock();
        }
…
};
```

*Use **SC_METHOD**:*

```
void grab_bus_method() {
   if (bus_access.trylock()) {
    /*access bus*/
   }
};
```

Một nhược điểm của sc_mutex là thiếu tín hiệu sự kiện/tín hiệu kích thích (event) khi mà **sc_mutex** được rảnh, khi mà yêu cầu sử dụng **trylock** nhiều lần căn cứ vào một vài event hay thời gian delay.

2.5.3. sc_semaphore

Một vài tài nguyên bạn cần có nhiều hơn một bản copy hay sở hữu. Để quản lý tài nguyên loại này, SystemC cung cấp **sc_semaphore**. Khi tạo **sc_semaphore**, nó cần xác định có sẵn bao nhiêu . Theo nghĩa này, mutex là semaphore với biến đếm bằng 1.

```
sc_semaphore        NAME (count);
NAME.wait();
NAME.trywait();
NAME.get_value();
NAME.post();
```

Điều quan trọng cần nhận thấy là **sc_semaphore::wait** là phương thức khác với hàm **wait().** Thực tế là, **sc_semaphore**::**wait**() thực thi giống với lệnh **wait(event).**

Ví dụ:

```
SC_MODULE(gas_station) {
   sc_semaphore pump(12);
   void customer1_thread {
        for(;;) {
        // wait until tank emty
```

```
        …
        // find an available gas pump
        pump.wait(); //fill tank & pay
    }
};
```

Ví dụ: sử class với muti-port

```
class multiport_RAM {
    sc_semaphore read_ports(3);
    sc_semaphore write_ports(2);
    …
    void read(int addr, int& data) {
        read_ports.wait();      // đọc vào
        read_ports.post();
    }
    void write(int addr, int data) {
        write_ports.lock();     // đọc vào
        write_ports.unlock();
    }
…
};
```

2.5.4. sc_fifo

sc_fifo được sử dụng phổ biến nhất cho quá trình mô phỏng. First-in First-out (FIFOs) là cấu trúc chung được sử dụng để quản lý luồng dữ liệu. FIFOs là một trong những cấu trúc đơn giản nhất để quản lý dữ liệu.

Về mặc định**, sc_fifo** có chiều dài (depth) là 16. **Sc_fifo** có thể chứa bất kỳ kiểu dữ liệu nào bao gồm lớn và cấu trúc phức tạp.

Ví dụ:

Sc_fifo *<ELEMENT_TYPENAME> NAME(SIZE);*

*NAME.**write** (VALUE);*

*NAME.**read**(REFERENCE);*

*…=NAME.**read()***

***if** (NAME.**nb_read**(REFERENCE)) { //non-blocking, true if success*

….

}

***if** (NAME.**num_available**()==0)*

*wait(NAME.**data_written_event**());*

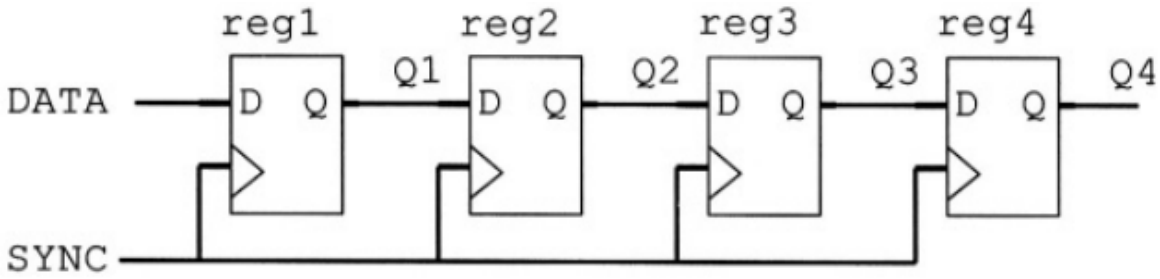
***if** (NAME.**num_free**()==0)*

***next_trigger**(NAME.**data_read_event**());*

Theo ví dụ, FIFOs thường được sử dụng làm dữ liệu đệm. Các giao thức truyền nối tiếp hay song song đều cần bộ đệm để lưu trữ dữ liệuS tạm thời. Trạng thái bộ đệm sẽ được cập nhật liên tục để hệ thống đọc hoặc ghi dữ liệu nhịp nhàng.

2.6. Đánh giá – Update channel

Chương này giới thiệu cách thức hoạt động của các channel để giúp chúng ta mô phỏng được hành vi song song của phần cứng. Chúng ta sẽ xem xét ví dụ về một thanh ghi dịch:



DATA đi từ trái sang phải đồng bộ theo xung clock SYNC. Trong software (ví dụ như C/C++), điều này được mô phỏng với bốn phép gán theo thứ tự như sau:

Q4 = Q3

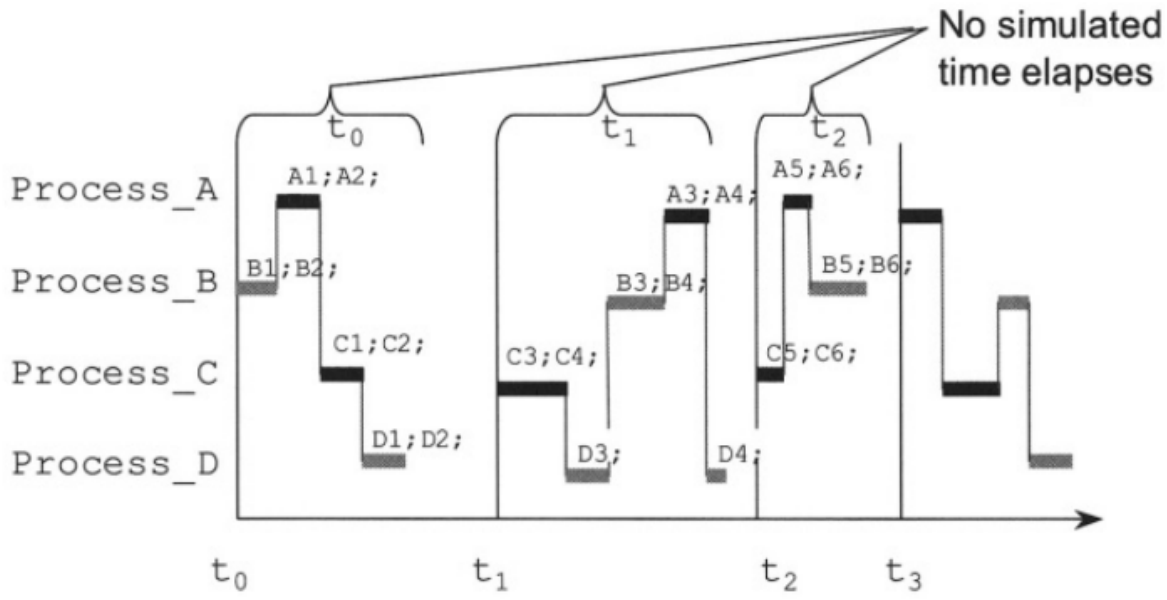
Q3 = Q2

Q2 = Q1

Q1 = DATA

Với cách làm việc như thế thì thứ tự rất quan trọng. Trong hardware thì mọi thứ phức tạp hơn. Mỗi thanh ghi (reg1…reg4) là một process chạy đồng thời và không độc lập với nhau. Simulator sắp xếp các process không theo một thứ tự, nghĩa là ta không biết thứ tự chạy

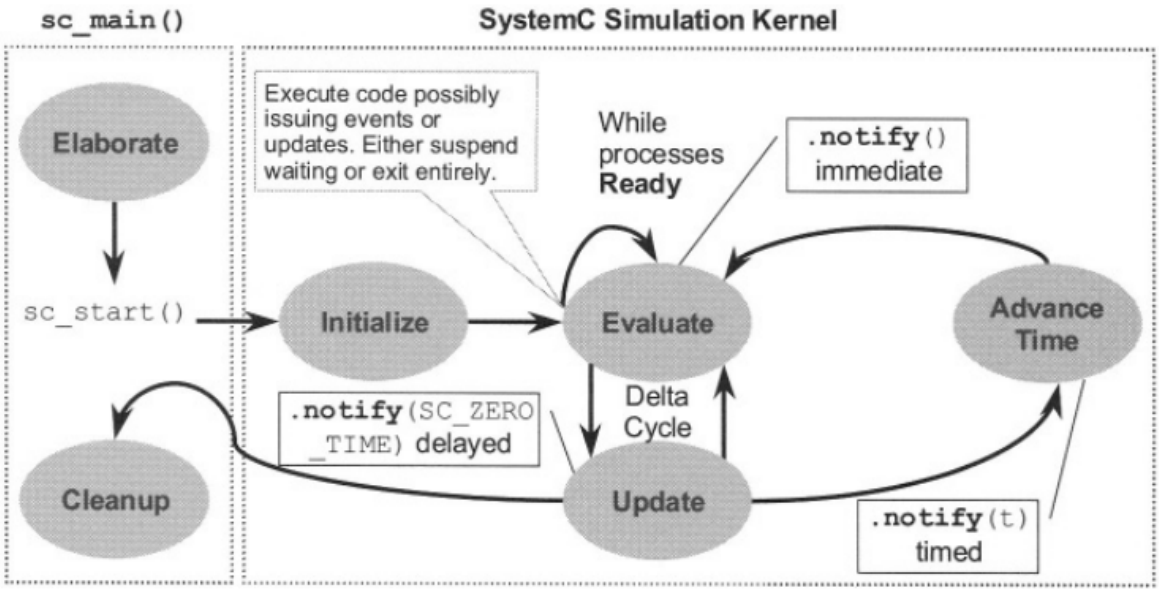
của các process. Hình minh họa dưới mỗi process đại diện cho một thanh ghi:



Vì không có gì đảm bảo rằng process nào sẽ chạy trước nên SystemC đưa ra ý tưởng sử dụng event để tác động đế thứ tự của các proces.

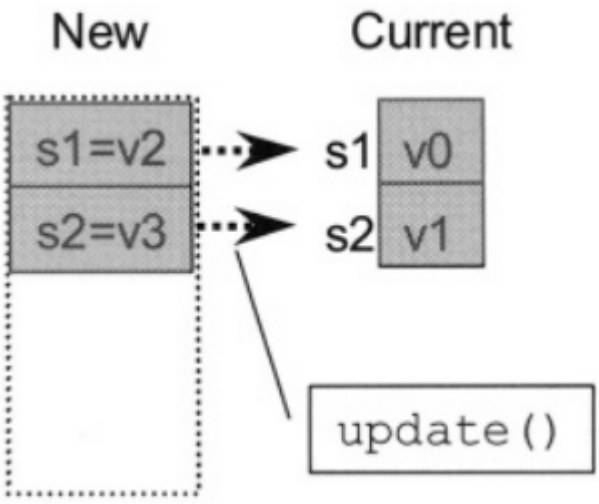
2.6.1. Cơ chế mô phỏng hành vi song song của SystemC kernel

Simulator của SystemC sử dụng mô hình Evaluate – Update.



Simulation kernel của SystemC có 2 phase chính là Evaluate và Update. Hai phase có thể chuyển đổi qua lại liên tục với nhau. Một sự chuyển đổi từ Evaluate đến Update và về lại Evaluate được gọi là một delta-cycle. Thậm chí khi simulator đi từ Evaluate đến Update đến Advance time thì đó cũng là một delta-cycle.

Các channel signal sử dụng phase Update để đồng bộ data. Mỗi channel được cấp phát 2 vùng nhớ lưu trữ, một chứa giá trị data hiện tại và một chứa giá trị mới của data.



Khi một process ghi data vào channel signal tại phase Evaluate, process đưa giá trị đó vào vùng nhớ dành cho giá trị data mới và gọi phương thức **request_update()** nói cho simulation kernel biết để simulation kernel gọi phương thức **update()** của channel trong phase Update.

Sau khi phase Evaluate hoàn thành, kernel gọi phương thức **update()** cho mỗi thể hiện channel để yêu cầu một sự cập nhật giá trị mới. Phương thức **update()** ngoài việc sao chép giá trị mới vào nơi lưu trữ giá trị cũ, nó còn có thể giải quyết một thông báo của sc_event (Ví dụ như có thể xác định một sự thay đổi nào đó và đánh thức một process đang trong trạng thái chờ).

Một chú ý rất quan trong là **data của channel không được cập nhật tại phase Evaluate mà được cập nhật tại phase Update**. Nếu một process ghi data vào một channel có cơ chế Evaluate-Update và sau đó ngay lập tức đọc data của channel thì sẽ thấy giá trị data vẫn như cũ không thay đổi.

2.6.2. sc_signal, sc_buffer

sc_signal<> và sc_buffer<> là hai channel sử dụng cơ chế Evaluate-Update. Cú pháp khai báo, đọc data từ channel và ghi data vào channel:

sc_signal<*datatype*> *signame* ;

signame.**write**(*newvalue*);

signame.**read**(*varname*);

// occurred in previous delta-cycle

sc_signal<> và sc_buffer<> chỉ cho phép một process duy nhất ghi giá trị vào nó trong suốt một delta-cycle. Điều này giúp ta tránh được sự nguy hiểm có thể gây ra crash hệ thống khi 2 hay nhiều process cùng lúc ghi những giá trị khác nhau vào tạo ra hiện tượng tranh chấp dữ liệu (race condition).

2.6.3. sc_signal_resolved, sc_signal_rv

Hai channel được dùng khi data có thể có rơi vào trường hợp tổng trở cao (Z) hoặc hở mạch (X).

sc_signal_resolved *name;*

sc_signal_rv<*WIDTH*> *name;*

2.6.4. Các template specialization của channel sc_signal

sc_signal có một biến template là **typename**. Systemc định nghĩa thêm một vài hành cho sc_signal<bool> mà các sc_signal<datatype> khác không có các hành vi này, ví dụ như **posedge_event()** và **negedge_event():**

sensitive << *signame.**posedge_event()***

<< *signame.**negedge_event()**;*

wait(*signame.**posedge_event()***

|*signame.**negedge_event()**);*

if (*signame.**posedge_event()***

|*signame.**negedge_event()**) {*

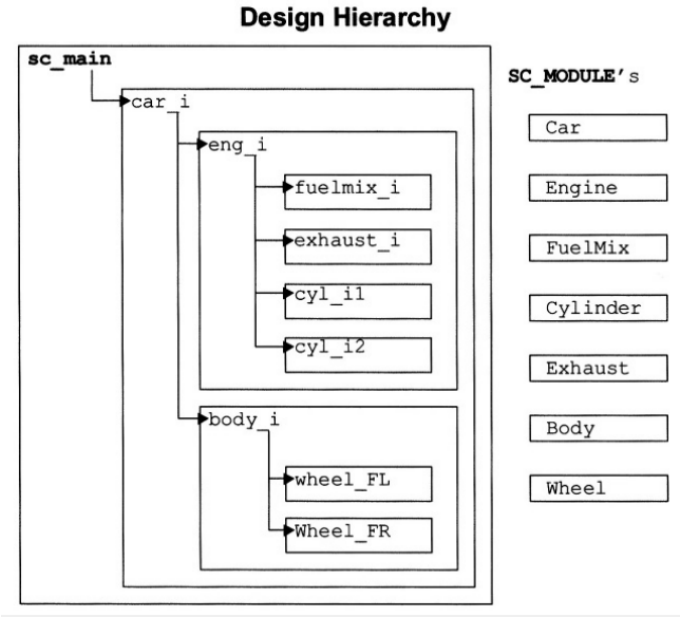
Chú ý rằng sc_buffer không được hỗ trợ các template specialization này.

2.7. Hệ thống phân cấp thiết kế

2.7.1. Hệ thống phân cấp module

Đối với các hệ thống lớn đòi hỏi phải phân vùng và phân cấp để dễ thiết kế và quản lý. Thiết kế hệ thống phân cấp trong SystemC là sử dụng các thể hiên của module con như là

dữ liệu thành viên của module cha. Nói cách khác, để tạo một mức đọ phân cấp, chúng ta tạo ra một đối tượng sc_module đại diện cho module cấp thấp hơn trong module cha. Ví dụ:



Trong ví dụ này, ta có một module cha đặt tên là **Car** và một module con tên là **Engine**. Để có được các mối quan hệ thữ bậc, chúng ta tạo ra một đối tượng **Engine** và một đối tượng **Body** trong định nghĩa lớp **Car**.

Các khởi tạo module xảy ra bên trong hàm dựng. chúng ta có thể định nghĩa phần code thực hiện hàm dựng trong file header hoặc trong file thực thi.

C++ cung cấp hai cách thức cơ bản để khởi tạo một module. Cách thứ nhất là một đối tượng module có thể được khởi tạo trực tiếp bằng cách khai báo. Ngoài ra, một đối tượng module có thể được tham chiếu gián tiếp bằng con trỏ và cấp phát động.

Tạo ra một hệ thống phân cấp tại top level (sc_main) thì hơi khác so với khi khởi tạo trong module. Sự khác biệt chủ yếu là về yêu cầu cú pháp C++ để khởi tạo bên ngoài một định nghĩa lớp.

Có 6 phương pháp phân cấp:

- Direct top-level
- Indirect top-level
- Direct sub-module header-only

- Direct sub-module
- Indirect sub-module header-only
- Indirect sub-module

2.7.2. Direct top-level

Phương pháp này đơn giản. Các design con được khai báo và khởi tạo trong 1 file thiết kế. Ví dụ:

```
//FILE: main.cpp

#include "Wheel.h"

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

Wheel wheel_FL("wheel_FL");

Wheel wheel_FR("wheel_FR");

sc_start();

}
```

2.7.3. Indirect top-level

Một thay đổi nhỏ trong cách tiếp cận, đặt thiết kế vào vùng nhớ heap bằng cách khai báo hai con trỏ và cấp phát một vùng nhớ động cho hai con trỏ bằng toán tử **new.** Việc thay đổi này làm chương trình của ta trông có vẻ dài hơn, tuy nhiên giúp thiết kế có khả năng cấu hình động. Ví dụ:

```
//FILE: main.cpp

#include "Wheel.h"

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

Wheel* wheel_FL; // con trỏ trỏ đến FL wheel

Wheel* wheel_FR; // con trỏ trỏ đến FR wheel

wheel_FL = new Wheel ("wheel_FL"); // tạo ra FL

wheel_FR = new Wheel ("wheel_FR"); // tạo ra FR

sc_start();
```

```
delete wheel_FL;

delete wheel_FR;

}
```

2.7.4. Direct sub-module header-only

Khi làm việc với một module con (ví dụ như bên dưới hoặc bên trong một module), mọi thứ trở nên nhẹ nhàng và thú vị hơn vì sematic của C++ yêu cầu sử dụng một danh sách khởi tạo cho cách tiếp cận trực tiếp. Ví dụ:

```
//FILE:Body.h

#include "Wheel.h"

SC_MODULE(Body) {

Wheel wheel_FL;

Wheel wheel_FR;

SC_CTOR(Body)

    : wheel_FL("wheel_FL"), //khởi tạo

    wheel_FR("wheel_FR") //khởi tạo

{

// các khởi tạo khác

}

};
```

2.7.5. Indirect sub-module header-only

Không có ưu điểm rõ ràng.

```
//FILE:Body.h

#include "Wheel.h"

SC_MODULE(Body) {
```

```
Wheel* wheel_FL;

Wheel* wheel_FR;

SC_CTOR(Body) {

wheel_FL = new Wheel("wheel_FL");

wheel_FR = new Wheel("wheel_FR");

// other initialization

}

};
```

2.7.6. Direct sub-module

Một nhược điểm của phương pháp trước là sự phức tạp trong thân hàm dựng. Phương pháp này của chúng ta đưa hàm dựng vào file thực thi (file .cpp), và nếu trong hàm dựng có process thì ta phải sử dụng thêm macro SC_HAS_PROCESS. Ví dụ:

```
//FILE:Body.h

#include "Wheel.h"

SC_MODULE(Body) {

Wheel wheel_FL;

Wheel wheel_FR;

SC_HAS_PROCESS(Body);

Body(sc_module_name nm);

};

//FILE: Body.cpp

#include "Body.h"

// Constructor

Body::Body (sc_module_name nm)
```

```
: wheel_FL("wheel_FL"),
wheel_FR("wheel_FR"),
sc_module(nm)
{
// other initialization
}
```

2.7.7. Indirect sub-module

```
//FILE:Body.h
struct Wheel;
SC_MODULE(Body) {
Wheel* wheel_FL;
Wheel* wheel_FR;
SC_HAS_PROCESS(Body);
Body(sc_module_name nm); // Constructor
};
//FILE: Body.cpp
#include "Wheel.h"
// Constructor
Body::Body(sc_module_name nm)
: sc_module(nm)
{
wheel_FL = new Wheel("wheel_FL");
wheel_FR = new Wheel("wheel_FR");
```

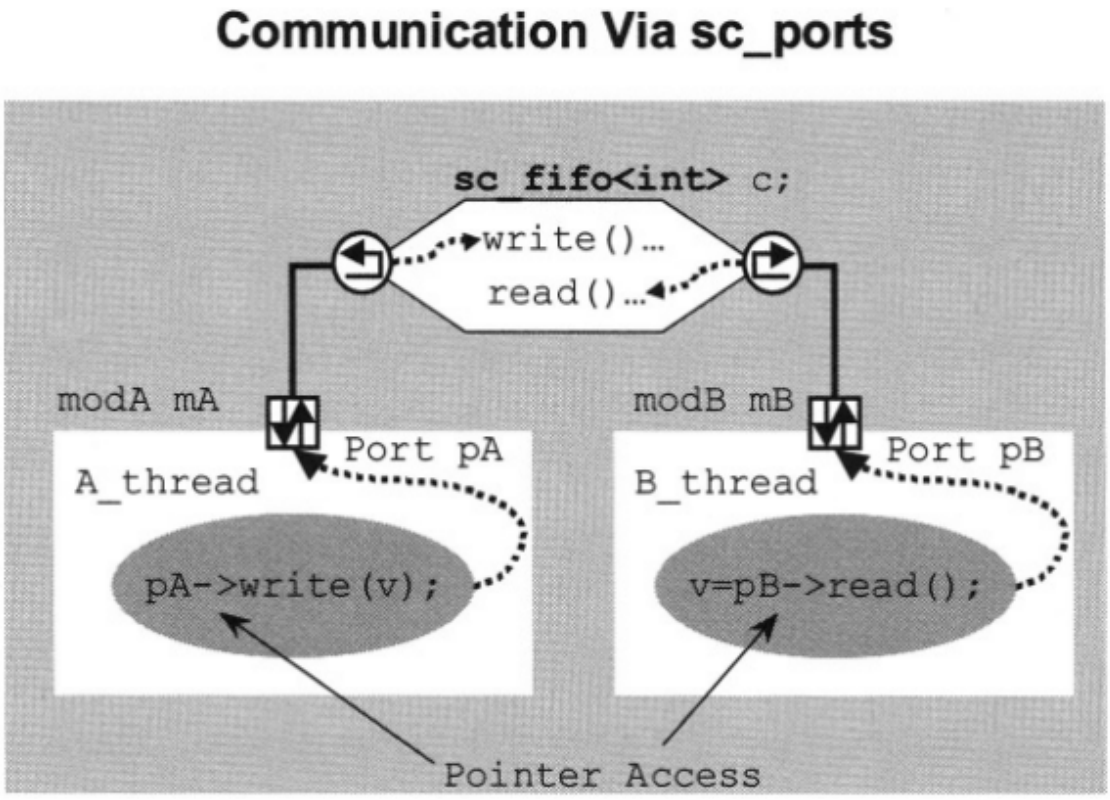
```
// other initialization
}
```

2.8. Giao tiếp

2.8.1. Port

Hệ thống phân cấp mà không có khả năng giao tiếp giữa các module với nhau thì không thể sử dụng. Khi giao tiếp giữa các module có 2 điểm cần lưu ý là "an toàn" và dễ sử dụng. "An toàn" ở đây có nghĩa là phải điều khiển được luồng giao tiếp để tránh trường hợp chạy đua giữa các process với nhau. Event và channel được sử dụng để xử lý vấn đề này.

Vấn đề dễ sử dụng thì khó trình bày hơn. Phương pháp giao tiếp của SystemC là SystemC sử dụng các channel chèn vào giữa các module cần giao tiếp với nhau. Có thể hiểu SystemC **dùng channel làm trung gian ở giữa** để giúp hai module có thể trao đổi, giao tiếp với nhau. Và SystemC cung cấp một khái niệm gọi là **port**. Port là một con trỏ trỏ đến một channel bên ngoài module, hiểu một cách nôm na thì port chính là cửa ngõ của module để module có thể giao tiếp được với bên ngoải. Ví dụ:

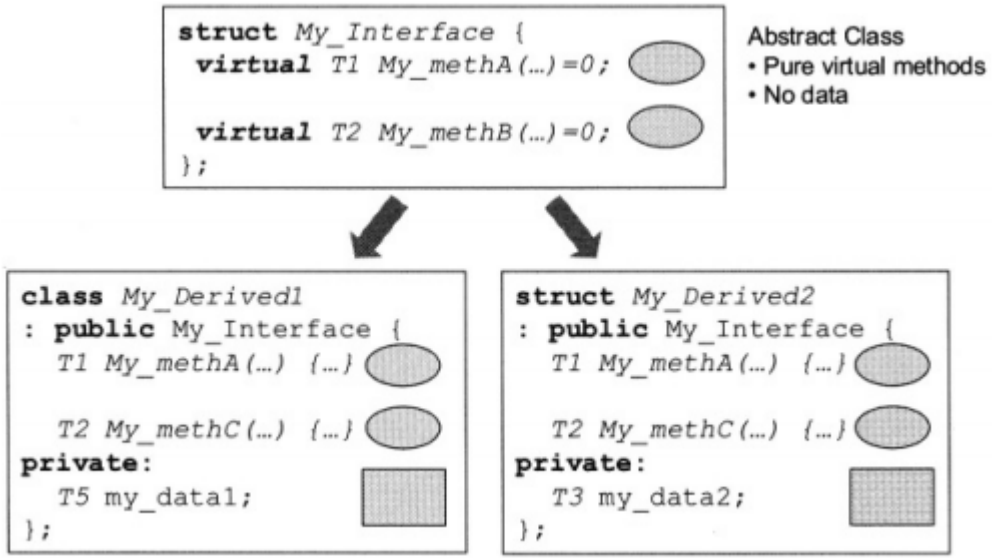


Ta có một module cha bên trong chứa một đối tượng module của **modA**, một đối tượng module của **modB** và một đối đối tượng channel **c** của sc_fifo. Process A_thread trong module **modA** truyền giá trị của một biến global **v** bằng cách gọi phương thức **write** của channel **c**. Process B_thread trong module modB có thể lấy giá trị đó bằng phương thức **read** cũng của channel **c**.

Chú ý rằng việc truy cập được thực hiện thông qua 2 con trỏ **pA** và **pB**. Và cũng chú ý rằng 2 process thật sự chỉ cần truy cập đến phương thức **write** và **read**. Cụ thể, **modA** chỉ cần truy cập đến **write**, và **modB** chỉ cần truy cập đến **read**.

2.8.2. Interface: C++ và SystemC

C++ có một khái niệm là lớp trừu tượng. Một lớp trừu tượng là một lớp không bao giờ được sử dụng một cách trực tiếp, mà chỉ được sử dụng thông qua lớp dẫn xuất. Lớp trừu tượng luôn chứa hàm thuần ảo. Hàm thuần ảo không được phép thực thi trong lớp trừu tượng. Các lớp dẫn xuất thừa kế từ lớp trừu tượng phải định nghĩa lại sự thực thi của các hàm thuần ảo. Một hàm thuần ảo ở trước có từ khóa **virtual** và được gán giá trị bằng 0. Ví dụ:



Nếu một không chứa bất kỳ thuộc tính hay phương thức thông thường nào mà chỉ chứa phương thức thuần ảo thì lớp đó được gọi là lớp interface. Ví dụ:

class my_interface {

virtual void write (**unsigned** addr, **int** data) = 0;

virtual int read(**unsigned** addr) = 0;

```
};
```

Nếu một đối tượng được khai báo như là một con trỏ đến một lớp interface, nó có thể được sử dụng với những lớp đa dẫn xuất. Giả sử chúng ta định nghĩa 2 lớp dẫn xuất sau đây:

```
struct multiport_memory_arch: public my_interface {
        virtual void write(unsigned addr, int data) {
                mem[addr] = data;
        }// end write
        virtual int read(unsigned addr) ) {
        return mem[addr];
        }//end read
private:
        int mem[1024];
};


struct multiport_memory_RTL: public my_interface {
        virtual void write(unsigned addr, int data) {
                // complex details of RTL memory write
        }// end write
        virtual int read(unsigned addr) ) {
                // complex details of RTL memory read
        }// end read
private:
        // complex details of RTL memory storage
```

```
};
```

Và bây giờ giả có một đoạn code để truy cập vào 2 lớp dẫn xuất ở trên:

```
void memtest(my_interface mem) {
        // complex memory test
}
multiport_memory_arch fast;
multiport_memory_RTL slow;
memtest(fast);
memtest(slow);
```

Như ta thấy, sử dụng cùng một lớp interface nhưng cách thực thiện các phương thức thì khác nhau. Ta có thể nghĩ interface như là một **Application programing interface (API)** cho một tập hợp các lớp dẫn xuất. Khái niệm này của C++ giống với khái niệm mà SystemC xây dựng nên port.

Định nghĩa: Một interface trong SystemC là một lớp trừu tượng kế thừa từ **sc_interface** và chỉ cho phép khai báo các phương thức thuần ảo được tham chiều bởi port và channel, không có sự thực thi hay dữ liệu thành viên trong đó.

Một channel SystemC là một lớp thực thi một hay nhiều lớp interface Systemc và kế thừa từ **sc_channel** hoặc **sc_prim_channel**. Một channel thực thi tất cả các phương thức của các lớp interface được kế thừa.

Bằng cách sử dụng interface để kết nối tới channel, chúng ta có thể thực thi các module độc lập với các chi tiết thực thi của các channel giao tiếp.

2.8.3. Khai báo Port trong SystemC

Định nghĩa: Một port là một template class kế thừa từ một interface SystemC. Port cho phép các channel truy cập đến module thông qua các kết nối.

Khai báo: **sc_port**<*interface*> *portname*;

Port luôn được định nghĩa bên trong module. Ví dụ:

SC_MODULE(stereo_amp) {

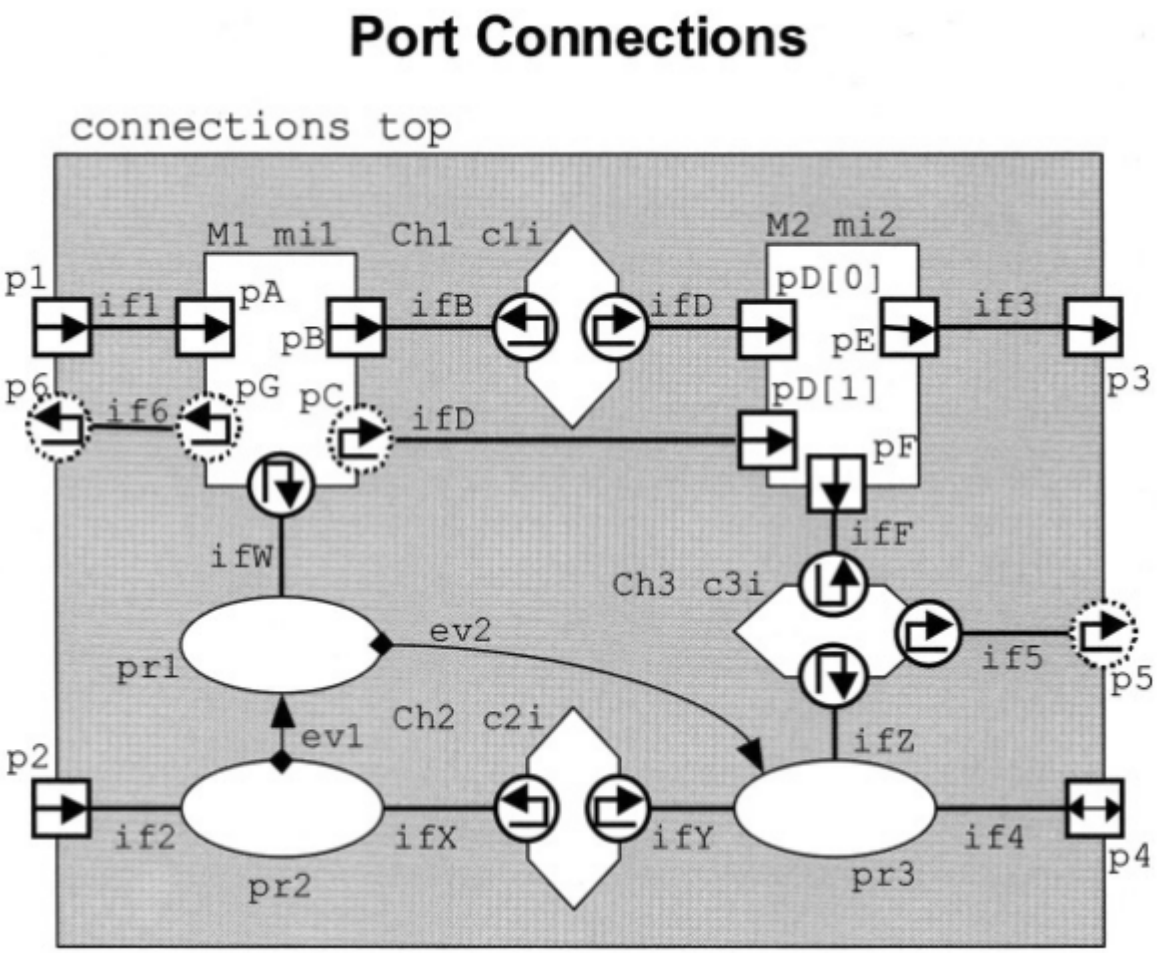
sc_port<**sc_fifo_in_if**<**int**> > soundin_p;

sc_port<**sc_fifo_out_if**<**int**> > soundout_p;

};

2.8.4. Các loại kết nối trong SystemC

Các kết nối có thể có trong SystemC là các kết nối giữa các port, các channel, các module và các process. Ví dụ



Ta có bảng tóm tắt các cách kết nối:

Đầu của kết nối	Cuối của kết nối	Phương thức

Port	Module con	Trực tiếp thông qua **sc_port**
Process	Port	Trực tiếp bằng process
Module con	Module con	Bằng channel
Process	Module con	Bằng channel hoặc thông qua **sc_export** hoặc interface được thực hiện bằng module con
Process	Process	Bằng event hoặc channel
Port	Channel	Trực tiếp thông qua **sc_export**

2.8.5. Kết nối Port

Các module được kết nối đến channel sau khi module và channel đã được tạo ra. Có 2 cú pháp dùng để kết nối đến port là dùng tên port và vị trí của port:

mod_inst.portname(channel_instance); // bằng tên

mod_instance(channel_instance,...); // bằng vị trí

2.9. Nhiều hơn về Port

2.9.1. Interface của sc_fifo

sc_fifo<> có 2 interface **sc_fifo_in_if**<> và **sc_fifo_out_if**>, cả hai đều có thể thực thi các phương thức mà **sc_fifo**<> có thể thực hiện. Trong thực tế các interface được định nghĩa trước để tạo ra các channel. Channel đơn giản chỉ là nơi thực thi các interface và giữ data.

sc_fifo_out_if<> cung cấp tất cả các phương thức cho việc xuất dữ liệu từ module đến **sc_fifo**<>. Module sử dụng phương thức **write()** hoặc **nb_write()** để đẩy dữ liệu lên FIFO.

sc_fifo_in_if() cung cấp tất cả các phương thức cho việc module nhận dữ liệu từ một **sc_fifo**<>. Module dử dụng phương thức **read()** hoặc **nb_read()** để nhận dứ liệu từ FIFO.

2.9.2. Interface của sc_signal

sc_signal<> có 3 interface **sc_signal_in_if**<>, **sc_signal_out_if**<> và **sc_signal_inout_if**. Cả 3 interface này đều có thể thực thi các phương thức của **sc_signal**<>. Và các interface này được định nghĩa trước khi channel được tạo ra. Channel đơn giản là nơi thực thi interface và thực thi hành vi **request-update** cho một tín hiệu.

2.9.3. sc_mutex và sc_semaphore

Hai channel **sc_mutex** và **sc_semaphore** cũng có các interface để sử dụng với port nhưng hỗ trợ kích thích bằng event, nếu muốn channel có kích thích bằng event ta phải tự tạo ra một channel cho riêng mình.

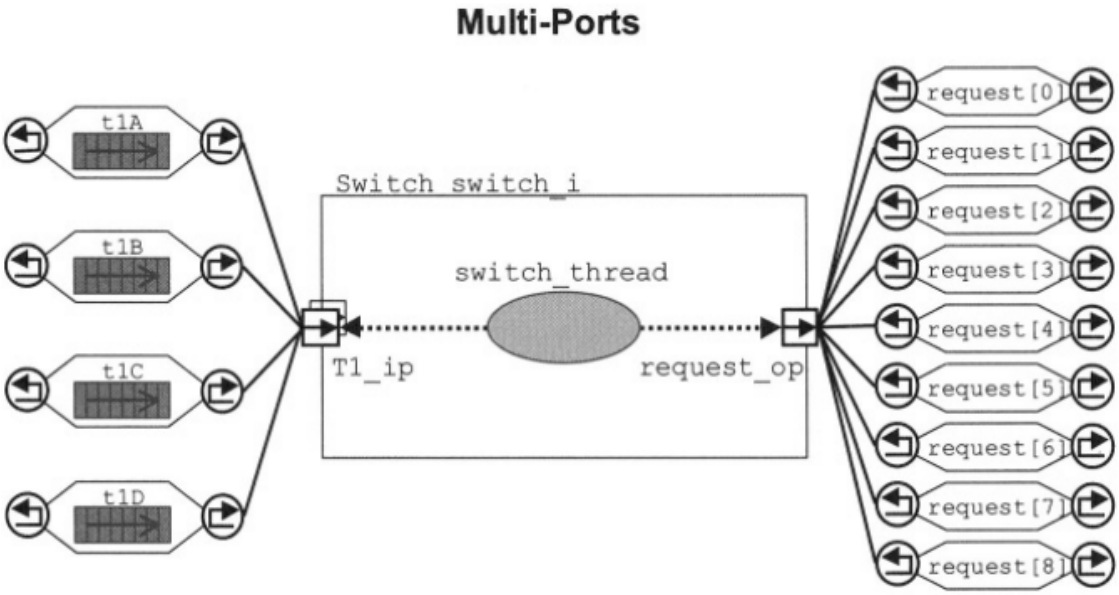
```
// Definition of sc_mutex_if interface

truct sc_mutex_if: virtual public sc_interface

 {

        virtual int lock() = 0;

        virtual int trylock() = 0;

        virtual int unlock() = 0;

};

// Definition of sc_semaphore_if interface

struct sc_semaphore_if : virtual public sc_interface

{

        virtual int wait() = 0;

        virtual int trywait() = 0;

        virtual int post() = 0;

        virtual int get_value() const = 0;

};
```

2.9.4. Mảng sc_port<>

sc_port<> có thêm một tham số thứ 2 là kích thước mảng. Tham số này dùng để tạo ra nhiều port có định nghĩa giống nhau.

sc_port <*interface [,N]*> *portname;* // **N = 0 . . MAX mặc định N = 1**

Với $N \neq 0$, phải có N channel kết nối đến port. Trường hợp đặc biệt N = 0 thì hầu như không có giới hạn về port. Nói cách khác, ta có thể kết nối bất một số lượng bất kỳ các channel tới port. Ví dụ:

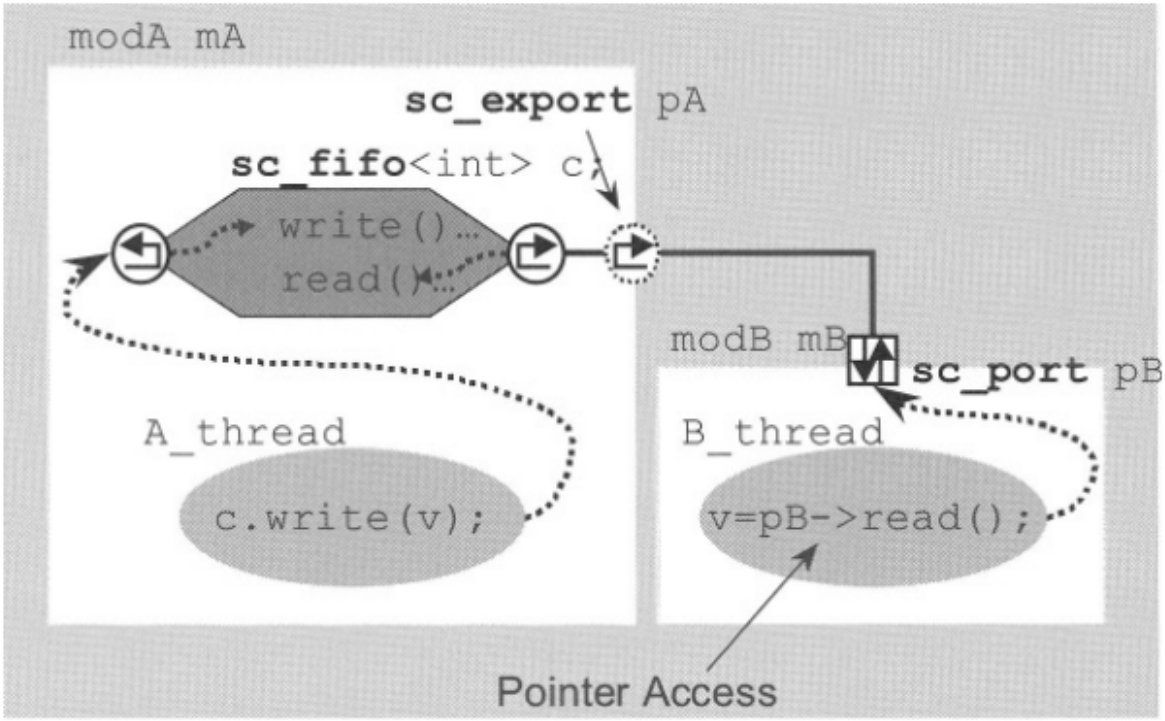



```
//FILE: Switch.h

SC_MODULE(Switch)

{

        sc_port<sc_fifo_in_if<int>, 4> T1_ip;

        sc_port<sc_signal_out_if<bool>,0> request_op;

};
```

Các channel được kết nối đến mảng port giống như cách các cổng bình thường được kết nối trừ các mảng port có nhiều hơn một kết nối. Trên thực tế, cú pháp của port cơ bản đơn giản là dựa vào trường hợp mặc định N = 1. Khi N > 1, mỗi kết nối được gán một vị trí trong mảng trên cơ sở **first-connected first-position**.

2.9.5. Export

SystemC có một loại port gọi là **sc_export**. Port này có cú pháp khai báo và định nghĩa giống như các port chuẩn của Systemc nhưng khác trong cách kết nối. **sc_export** là một chuyển kết bên trong một module, và dùng các port bên ngoài như là một channel. Ví dụ:



Cú pháp:

sc_export<*interface> portname;*

Kết nối tới một **sc_export** cần một số thay đổi:

SC_MODULE(*modulename*)

{

sc_export<*interface> portname;*

channel cinstance;

SC_CTOR(*modulename*)

{

portname (*cinstance*)*;*

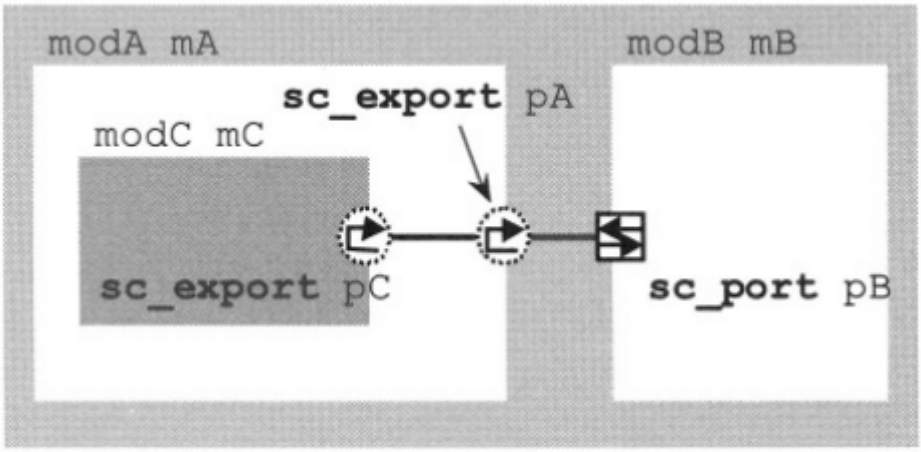
}

};

Ví dụ sau mô tả process chuyển đổi một tín hiệu theo chu kỳ:

```
SC_MODULE(clock_gen)
{
    sc_export<sc_ signal<bool> > clock_xp;
    sc_signal<bool> oscillator;
    SC_CTOR(clock_gen)
    {
        SC_METHOD(clock method);
        clock_xp(oscillator); // connect sc_signal
        // channel
        // to export clock_xp
        oscillator.write(false);
    }
    void clock_method()
    {
        oscillator.write(!oscillator.read());
        next_trigger(10,SC_NS);
    }
};
```

sc_export còn một ứng dụng mạnh nữa là interface trong hệ thống phân cấp thiết kế:

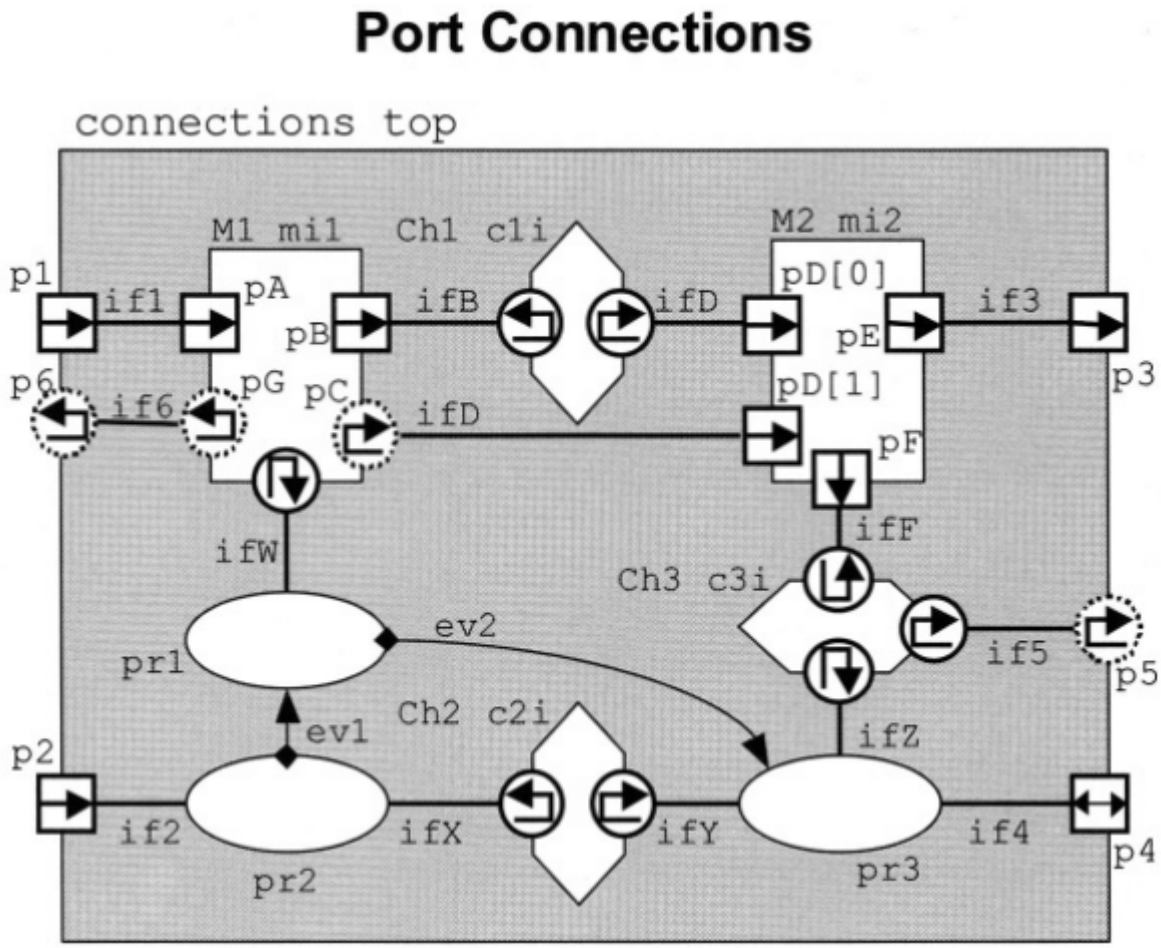


Giống như **sc_port**, **sc_export** có thể được kết nối trực tiếp đến một **sc_export** khác trong hệ thống phân cấp. Ví dụ:

SC_MODULE (*modulename*)
{
    sc_export<*interface*> *xportname;*
    module minstance;
    SC_CTOR(*modulename*), *minstance* (*"minstance"*)
    {
        xportname(*minstance.subxport*);
    }
};

Ta cũng có thể khai báo một mảng các **sc_export** như mảng **sc_port**.

2.9.6. Tổng kết các loại kết nối



Hình trên thể hiện tất cả các kết nối có thể có:

- Các process có thể giao tiếp với các process khác trong cùng một module sử dụng channel. Ví dụ: process **pr2** giao tiếp với process **pr3** thông qua interface **ifx** của channel **c2i**.
- Các process có thể giao tiếp với các process khác trong cùng một module sử dụng **event** để đồng bộ sự thay đổi của thông tin thông qua các biến data, Ví dụ: process **pr2** giao tiếp với process **pr1** thông qua even **ev1**.
- Các process có thể giao tiếp với các process ở cấp cao hơn trong hệ thống phân cấp sử dụng interface được truy cập thông qua sc_port. Ví dụ: process **pr2** thông qua port **p2** sử dụng interface **if2**.
- Các process có thể giao tiếp với các process bên trong các module con thông qua interface với channel được kết nối đến port của module con. Ví dụ: process **pr3** kết nối đến module **mi2** thông qua interface **ifz** của channel **c3i**.

- **sc_export** có thể kết nối thông qua interface đến channel. Ví dụ: port **p5** kết nối đến channel **c3i** sử dụng interface **if5**.
- **sc_export** có thể kết nối trực tiếp đến sc_export của module con. Ví dụ: port **p6** kết nối trực tiếp đến port **pG** của module con **mi1**.
- **sc_port** có thể kết nối gián tiếp đến các process bằng cách để cho các process truy cập interface. Cũng có nghĩa là process truy cập đến port. Ví dụ: process **pr1** giao tiếp với module con **mi1** thông qua interface **ifw**.
- **Mảng sc_port** có thể được sử dụng để tạo ra multiple port sử dụng cùng một interface. Ví dụ: **pD[0]** và **pD[1]** của module con **mi2** là một mảng port.

3. SERIAL COMMUNICATION INTERFACE (SCI)

3.1. Tổng quan về SCI

SCI là một thiết bị giao tiếp kỹ thuật số cho phép truyền nhận dữ liệu giữa CPU và các thiết bị ngoại vi không đồng bộ khác sử dụng chuẩn non-return-to-zero (NRZ) theo dạng serial (một bit tại một thời điểm trên đường truyền).

Đặc điểm kỹ thuật của SCI:

- **Chế độ giao tiếp:** đồng bộ và bất đồng bộ.
- **Phương thức giao tiếp:** Song công. Bộ truyền và nhận độc lập cho phép truyền và nhận dữ liệu liên tục. Việc truyền nhận dữ liệu liên tục có ở cả 2 phía truyền và nhận nhờ cấu trúc buffer kép..
- **Tốc độ bit:** bộ tạo tốc độ bit trên chip cho phép người dùng chọn tốc độ bit phù hợp. Có thể sử dụng nguồn clock bên ngoài như clock để truyền/nhận.
- **Định dạng dữ liệu:**
  - Có thể chọn giữa hai chế độ LSB-first và MSB-first (trừ trường hợp 7-bit dữ liệu ở chế độ bất đồng bộ.
  - **Chế độ bất đồng bộ:**
    - ✓ Độ dài dữ liệu: 7-bit hoặc 8-bit.
    - ✓ Chiều dài stoop bit: 1-bit hoặc 2-bit.
    - ✓ Parity: parity chẵn, parity lẻ hoặc không có parity.
  - **Chế độ đồng bộ:**
    - ✓ Độ dài dữ liệu: 8-bit.
- **Phát hiện lỗi nhận:**
  - **Chế độ đồng bộ:** Lỗi parity, lỗi overrun và lỗi frame.
  - **Chế độ đồng bộ:** Lỗi overrun.

3.1.1. Danh sách thanh ghi

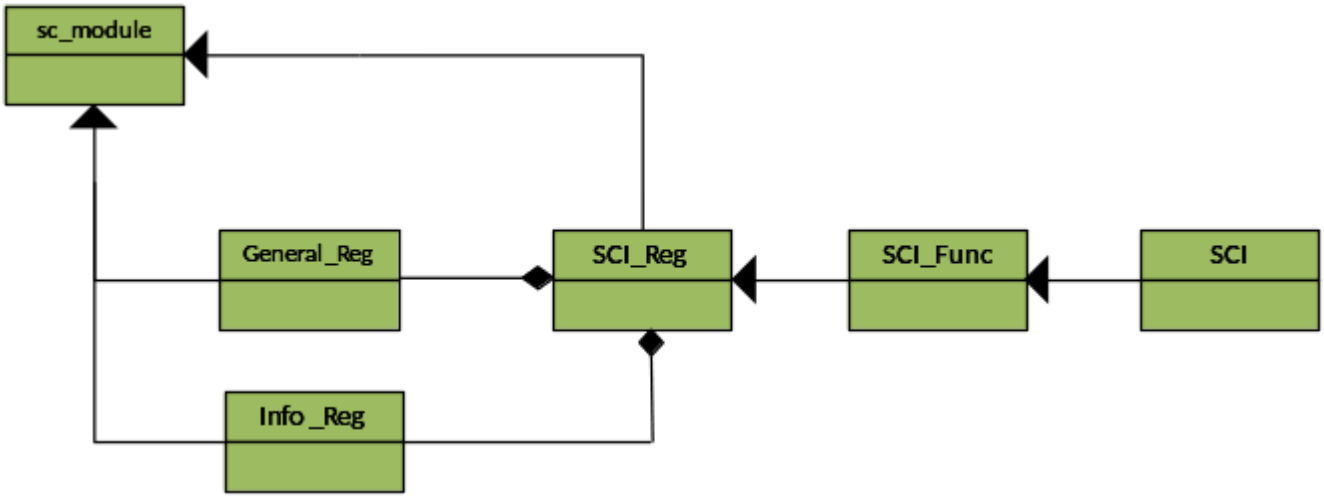
Thanh ghi	*Giá trị mặc định*	*Bit*	*Tên bit*	*Mô tả*	*Read/Write*
SCiCR 8 bit	H'00	[7]	TIE	Cho phép ngắt khi bộ buffer truyền trống (***khi bit TIE được set lên 1, bit TEIE phải được set xuống 0)***	R/W
		[6]	RIE	Cho phép ngắt khi bộ nhận đầy	R/W
		[5]	TE	Cho phép truyền	R/W
		[4]	RE	Cho phép nhận	R/W
		[3]	-	Reserved	R
		[2]	TEIE	Cho phép ngắt khi kết thúc truyền ***(khi bit TIE được set lên 1, bit TEIE phải được set xuống 0)***	R/W
		[1:0]	CKS	Chọn clock ***(chỉ được set khi TE = RE = 0)***	R/W
SCiMD 8 bit ****Nên set khi giá trị của bit TE và RE trong thanh ghi SCiCR là 0***	H'00	[7]	SMS	Chọn mode	R/W
		[6]	SDLS	Chọn độ dài Data	R/W
		[5]	PE	Cho phép parity	R/W
		[4]	OES	Chọn parity	R/W
		[3]	TSTLS	Chọn độ dài bit stop	R/W
		[2]	-	Reserved	R
		[1:0]	SCSS	Chọn nguồn đếm	R/W
SCiEMD 8 bit **** Nên set khi giá trị của bit TE và RE trong thanh ghi SCiCR là 0***	H'00	[7]	-	Reserved	R
		[6]	-	Reserved	R
		[5]	-	Reserved	R
		[4]	-	Reserved	R
		[3]	SDIR	Chọn hướng Data ***(bit SDIR bit chỉ có hiệu lực khi định dạng transmit/receive là 8-bit data)***	R/W
		[2]	-	Reserved	R
		[1]	CKPOS	Select clock polarity	R/W
		[0]	CKPSH	Select clock phase	R/W
SCiBR 8 bit	H'FF	[7:0]	BR	Setting tốc độ bit	R/W
SCiTB **(không thể truy xuất trực tiếp)** 8 bit ****Cờ TBEF trong thanh ghi SCiSR phải được set lên 1 (không có data trong thanh ghi buffer truyền) trước khi ghi data truyền vào***	H'00	[7:0]	TBD	Giữ data để truyền	R/W

thanh ghi SCiTB					
SCiRB **(không thể truy xuất trực tiếp)** 8 bit ****Kiểm tra cờ RBEF trong thanh ghi SCiSR phải được set lên 1 (data tồn tại trong thanh ghi buffer nhận) trước khi đọc data trong thanh ghi SCiRB***	H'00	[7:0]	RBD	Giữ data để nhận	R
SCiSR 8 bit	H'00	[7]	TBEF	Cho biết data không tồn tại trong thanh ghi buffer truyền	R/(W)*
		[6]	RBFF	Cho biết data tồn tại trong thanh ghi buffer nhận	R/(W)*
		[5]	OREF	Cho biết đã xảy ra một lỗi overrun	R/(W)*
		[4]	FREF	Cho biết đã xảy ra một lỗi frame	R/(W)*
		[3]	PERF	Cho biết đã xảy ra một lỗi parity	R/(W)*
		[2]	TSEF	Cho biết data không tồn tại trong thanh ghi dịch truyền (đã truyền xong)	R
		[1]	-	Reserved	R
		[0]	-	Reserved	R

3.1.2. Danh sách các port

Tên Pin	*I/O*	*Type*	*Khởi tạo*	*Mô tả*
SCKi	I/O	bool	-	Chân clock input/output của SCI
RXDi	I	sc_dt:sc_bv<12>	"000000000000"	Chân nhận data của SCI
TXD	O	sc_dt:sc_bv<12>	"000000000000"	Chân truyền data của

3.1.3. Các lớp trong thiết kế SCI



Chú ý: Lớp gọi một lớp khác

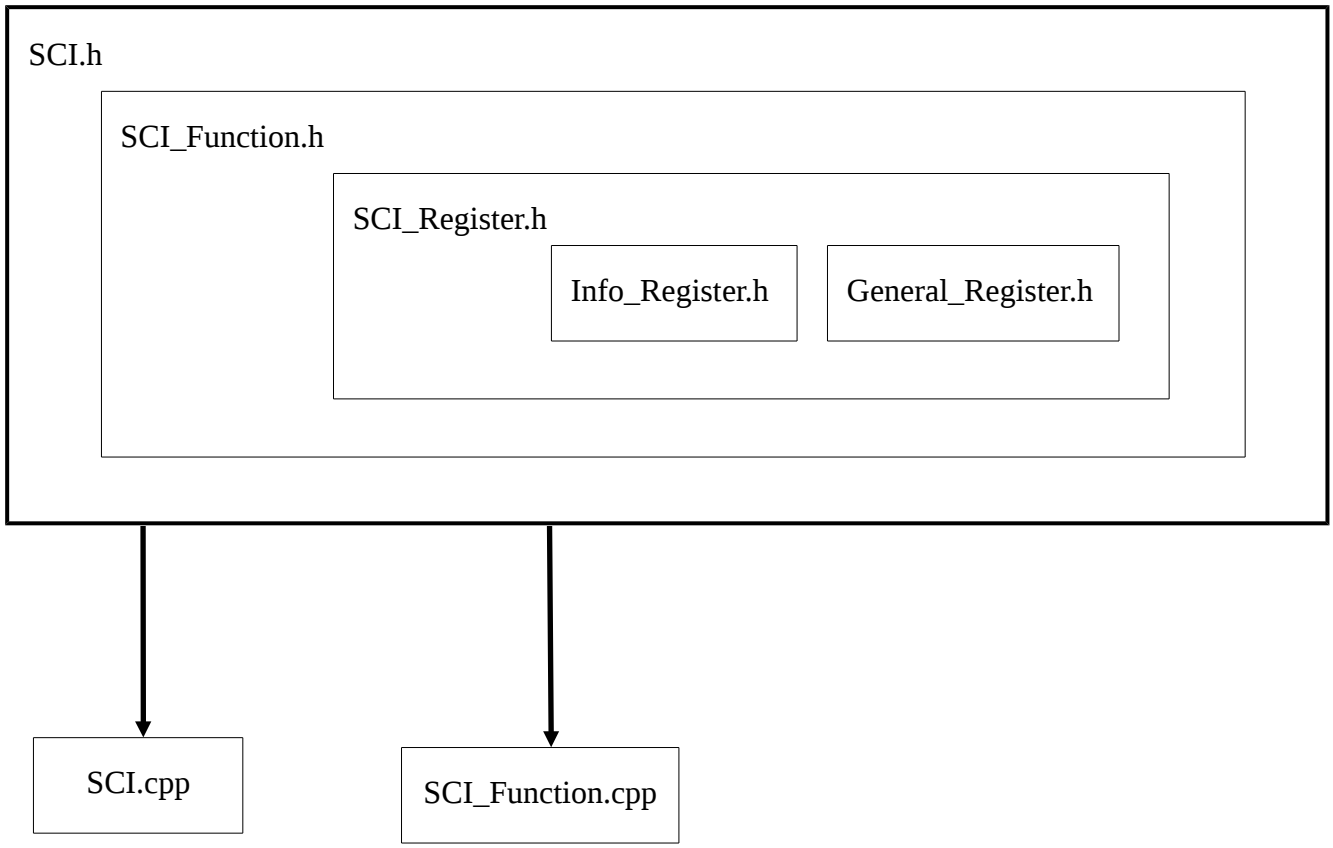
Kế thừa

No.	Tên lớp	Mô tả
1	sc_module	Lớp có sẵn trong thư viện SystemC, được dùng để thực hiện một module
2	Info_Reg	Lớp được sử dụng để lấy thông tin về thanh ghi: ➢ Giá trị khởi tạo của thanh ghi ➢ Số trường bit ➢ Tên trường bit ➢ Bit đầu của trường bit ➢ Bit cuối của trường bit ➢ Option của thanh ghi (chỉ đọc hoặc chỉ ghi hoặc vừa đọc vừa ghi)
3	General_Reg	Lớp được sử dụng để tạo ra 1 thanh ghi với đầy đủ đặc tính được lấy trong lớp Info_Reg
4	SCI_Reg	Lớp chứa các thanh ghi của SCI, kế thừa 2 lớp General_Reg và Info_Reg
5	SCI_Func	Lớp chứa các hàm chức năng của SCI
6	SCI	Lớp đại diện mô hình SCI, kế thừa lớp SCI_Func

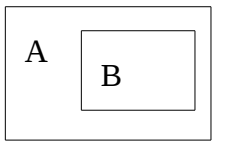
Info_Reg và General_Reg là 2 lớp được sử dụng để tạo ra 1 thanh ghi cụ thể. Lớp Info_Reg có 1 template xác định độ dài của thanh ghi (thanh ghi bao nhiêu bit) và một đối tượng của lớp Info_Reg nhận đối số là 1 file .txt có nội dung là sự mô tả các đặc điểm thanh ghi của (sẽ được trình bày chi tiết ở phần sau).

3.1.4. Mô tả thiết kế

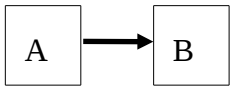
3.1.4.1. Cấu trúc file của thiết kế



Chú ý:



File A include file B



File A khai báo các intterface và được định nghĩa trong file B

STT	Tên file	Phát triển/Tái sử dụng	Mô tả
1	Info_Register.h	Tái sử dụng	Lớp Info_Reg tạo ra đối tượng chứa thông tin đặc tả thanh ghi
2	General_Register.h	Tái sử dụng	Lớp General_Reg tạo ra thanh ghi với đặc điểm được mô tả trong đối tượng tương ứng của lớp Info_Reg
3	SCI_Register.h	Phát triển	Lớp SCI_Reg chứa các đối tượng của 2 lớp Info_Reg và General_Reg, là các thanh ghi của SCI
4	SCI_Function.h	Phát triển	File header của lớp SCI_Func
5	SCI_Function.cpp	Phát triển	Thực hiện các hàm chức năng của SCI
6	SCI.h	Phát triển	File header của lớp SCI
7	SCI.cpp	Phát triển	Mô phỏng hành vi của SCI

3.1.4.2. File input

Để tạo ra một thanh ghi ta cần một file .txt mang đặc tả của thanh ghi cần tạo để truyền vào hàm dựng của đối tượng thuộc lớp Info_Reg. đối tượng đó sẽ xử lý các thông tin mô tả đó và lưu vào các thuộc tính public của mình và lớp Info_Reg cũng một template nhận một đối kiểu int quy định độ dài của thanhg ghi theo bit.

Cấu trúc của file:

- **Dòng đầu tiên** quy định giá trị khởi tạo mặc định của thanh ghi được biểu diễn dưới dạng bit.
- **Dòng thứ hai** là một số nguyên dương n quy định số trường bit trong thanh ghi.
- **n dòng tiếp theo** là đặc tả của mỗi trường bit, mỗi dòng có dạng:
  <Tên trường bit> [<bit start>:<bit end>] <Option>

  Trong đó: **<Option>** là mode của thanh ghi: **READ_ONLY**, **WRITE_ONLY**, **READ_WRITE** va **NONE**.

Thanh ghi trong thiết kế là một đối tượng của lớp General_Reg, hàm dựng đối tượng này sẽ lấy cácgiá trị của thuộc tính public của đối tượng tương ứng thuộc lớp Info_Reg và lưu lại trong các thuộc tính của mình. Tập thuộc tính của một đối tượng thuộc lớp General_Reg là tập thuộc tính của thanh ghi tương ứng. Tập thuộc tính này bao gồm: **reg_Default, bit_RangeNum, bit_Name, bit_Start, bit_End, bit_Option, reg_Data**. Các thuộc tính này được khai báo kiểu private để đảm bảo cấu trúc thanh ghi không bị truy cập trái phép bên ngoài làm hư hại thanh ghi. Do đó thuận lợi cho việc truy xuất thanh ghi, lớp General_Reg cung cấp các phương thức public như sau:

- **Ten_thanh_ghi.write(data)**: ghi dữ liệu vào thanh ghi.

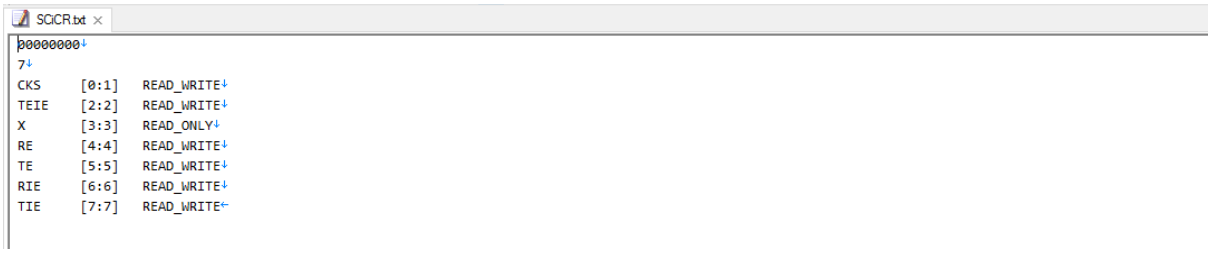
- **Ten_thanh_ghi.read()**: đọc data từ thanh ghi, phương thức này trả lại data đang có trong một thanh ghi theo dạng nhị phân.
- **Ten_thanh_ghi<do_dai_truong_bit>.write_bit("ten_truong_bit", data):** ghi data vào một trường bit của một thanh ghi.
- **Ten_thanh_ghi<do_dai_truong_bit>.read_bit("ten_truong_bit"):** đọc data từ một trường bit của một thanh ghi, phương thức này trả lại data đang có trong một trường bit của một thanh ghi theo dạng nhị phân.
- **Ten_thanh_ghi<do_dai_truong_bit>.read_bit_uint("ten_truong_bit"):** đọc data từ một trường bit của một thanh ghi, phương thức này trả lại data đang có trong một trường bit của một thanh ghi theo kiểu số nguyên thập phân không dấu.

Ví dụ ta muốn tạo ra một thanh ghi SCiCR có đặc điểm như sau:

7	6	5	4	3	2	1	0
TIE	RIE	TE	RE	-	TEIE	CKS	

Default: 00000000

Đầu tiên ta tạo một fileSCiCR.txt có nội dung như sau:



Sau đó ta tạo một đối tượng từ lớp Info_Reg nhận đối số là tên file vừa tạo. Tiếp tục tạo một đối tượng của lớp General_Reg nhận đối số là các thuộc tính public của đối tượng thuộc lớp Info_Reg vừa tạo:

```
Info_Reg<8> SCiCR_Info("SCiCR_Info", "SCiCR.txt");
General_Reg<8> SCiCR SCiCR("SCiCR", SCiCR_Info.reg_Default,
SCiCR_Info.bit_RangeNum,
SCiCR_Info.bit_Name, SCiCR_Info.bit_Start, SCiCR_Info.bit_End, SCiCR_Info.bit_Option);
```

Thanh ghi sau tạo xong sẽ được truy cập thông qua tên của đối tượng thuộc lớp General_Reg vừa tạo và các phương thức public mà lớp General_Reg cung cấp.