Module

Kun-Chih (Jimmy) Chen 陳坤志

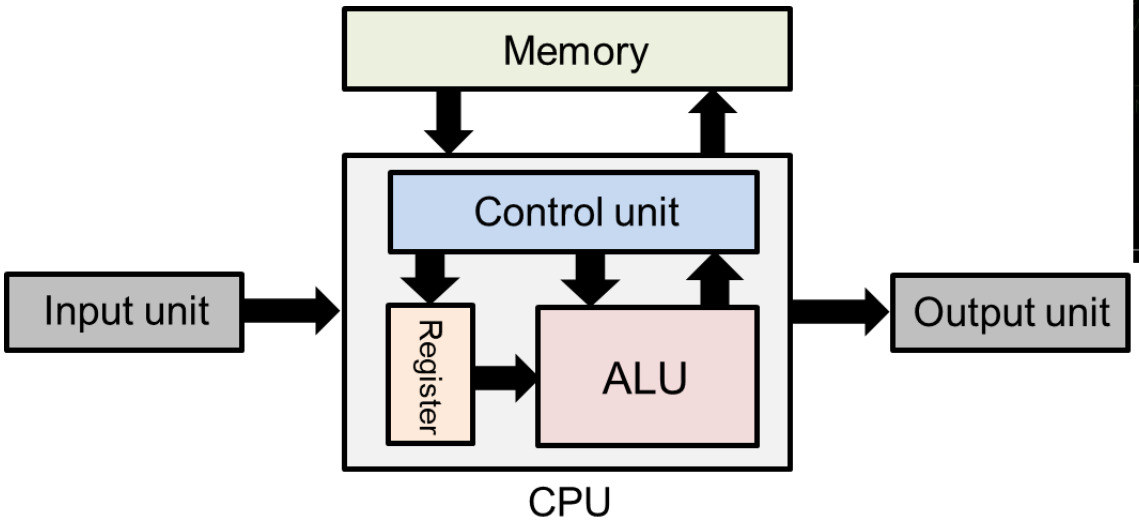
kcchen@nycu.edu.tw

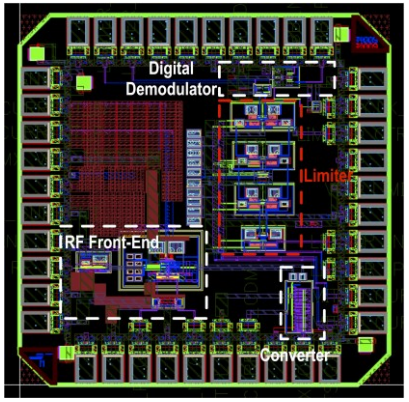
Institute of Electronics,
National Yang Ming Chiao Tung University

Module Overview

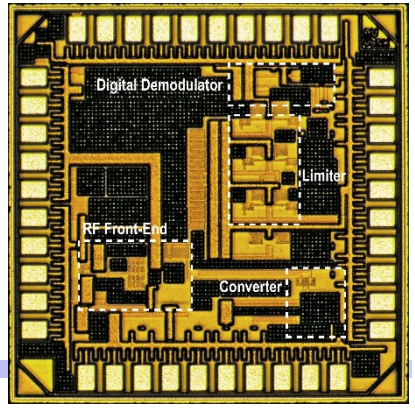
- Complex systems consist of many independently component.
  - A component can be represented a module
  - Component may be smaller or larger

- Allow designers to hide internal data representation from other modules.
  - Easier to maintain design





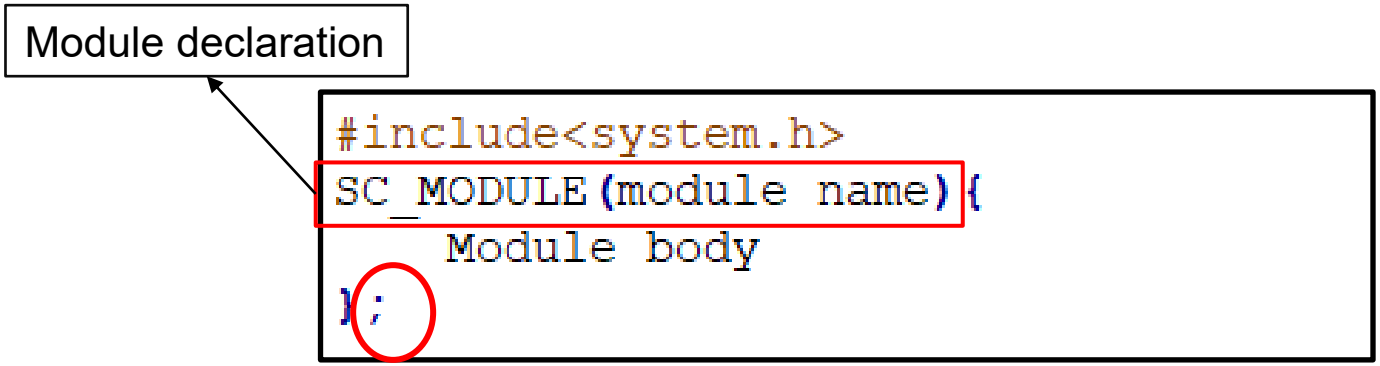
Chip layout



Micrograph

Module

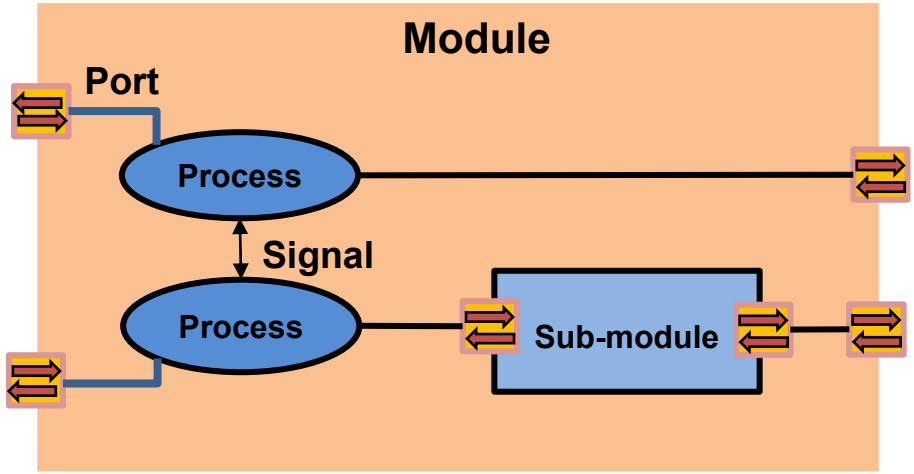
- **SC_MODULE** is the keyword that represents component in SystemC
- A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity.
- A SystemC module is simply a C++ class definition



Module Body

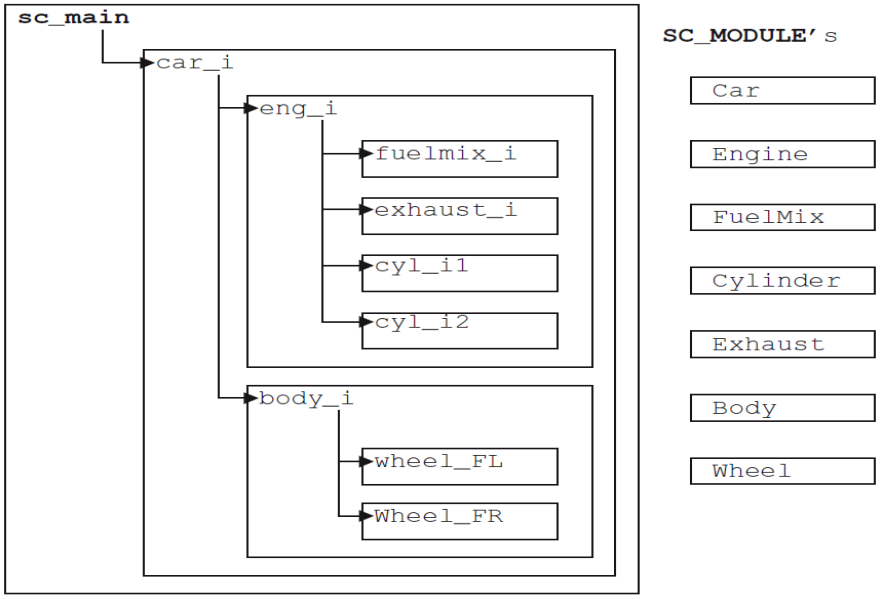
- A module is the basic structural building block in SystemC

- Module body
  - Ports
  - Member channel instances
  - Member module instances (sub-designs)
  - Constructor
  - Simulation process member functions (processes)



Module Hierarchy

- In small system design, a module represents a functionable component.
- Lager system designs require partitioning and hierarchy to enable project management
- A top module may include some sub-modules



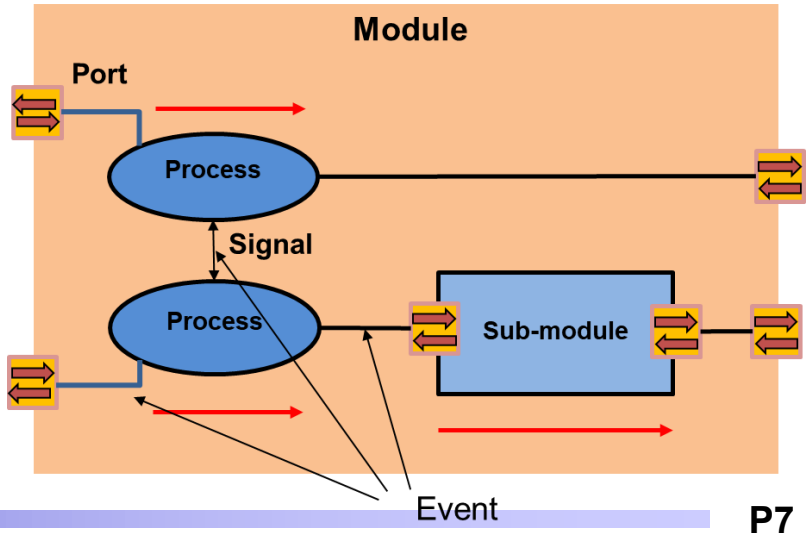
Module Hierarchy

- Using pointers and dynamic memory allocation

```
#include<submodule_name.h>
SC_MODULE(module_name){
    Module port declarations
    ...
    module_name_A *instance_name_A;
    module_name_N *instance_name_N;
    ...
    Local channel declarations

    SC_CTOR(module_name)
    {
        nstance_name_A = new submodule_name("instance_name");
        instance_name_A->subport_name(modport_name);
        instance_name_A->subport_name(local_channel_name);
        or:
        (*instance_name_N)(modport_name, local_channel_name,...);
        ...
    }

};
```

Parallel Processing

- Typically, methods in programming language are executed sequentially.
  - The function is called in order
- However, electronics systems are inherently parallel activities.
  - Each module are activated parallel
  - When the input signal change, the circuit will work
- Different from conventional C language, SystemC adopt the event-based simulation
  - Modules are activated by event



Main Function

- Program have initialization requirement in main function.
- In C/C++, the starting point is called **main()**.
- In SystemC, the starting point is called **sc_main()**.

```
int main(int argc, char* argv[]){
    BODY_OF_PROGRAM

    return EXIT CODE; //Zero indicates success
}
```

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

    ELABORATION
    sc_start();  // <-- Simulation begin & ends
                 // in this function
    [POST_PROCESSING]
    return EXIT CODE
}
```

Main Function

- Within **sc_main()**, code executes in three distinct major stages
  - Elaboration stage
  - Simulation stage
  - Post-processing stage
- During elaboration, connectivity for the model is established.
  - At the end of elaboration, **sc_start**() invokes the simulation stage
- During simulation, code representing the behavior of the model executes.
- Post-processing finishes with the return of an exit status from **sc_main**()

sc_start

- sc_start is a key method in SystemC
  - Start the simulation phase
  - Initialization and execution
- sc_start()
  - The scheduler will run until it completes
- sc_start(value, sc_time_unit)
  - The value is the execution time

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

    ELABORATION
    sc_start();  // <-- Simulation begin & ends
                 // in this function
    [POST_PROCESSING]
    return EXIT CODE
}
```

sc_main

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

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

    //port declaration
    sc_signal<int> sig_a, sig_b, sig_c;

    //module declaration
    Adder add("add");

    //signal connection
    add(sig_a,sig_b,sig_c);
    or
    add.sig_a(sig_a);
    add.sig_b(sig_b);
    add.sig_c(sig_c);

    //run simulation
    sc_start(100,SC_SEC);
    return 0;

}
```

main.cpp

Constructor in SystemC

- Since **SC_MODULE** is a C++ class in SystemC, it requires a constructor to perform several tasks.
  - Initializing / allocating sub-designs
  - Connecting sub-designs
  - Registering processes with the SystemC kernel
  - Providing static sensitivity
  - Miscellaneous user-defined setup

- To simplify coding, SystemC provides the macro, **SC_CTOR**()

Sub-design Allocation

- If the module include multiple sub-module, it is necessary to declare the sub-module in the **SC_CTOR**()
  - Create module hierarchy

- Using the pointer to create sub-modules

```
#include<submodule_name.h>

SC_MODULE(submodule_name){

    module_name_A *instance_A;
    module_name_N *instance_N;

    SC_CTOR(){
        (*instance_A)(port_name1, port_name2, ...);
        (*instance_N)(port_name1, port_name2, ...);
    }

};
```

Sub-design Connectivity

- Each port of sub-module need to be connected to the top module
- There are two syntax for connecting port
  - By name
  - By position

```
#include<submodule_name.h>

SC_MODULE(submodule_name){
    Module port declarations;

    module_name_A *instance_A;
    module_name_B *instance_B;

    SC_CTOR(module_name){
        (*instance_A)(port_name1,port_name2, ...);
        (*instance_B)(port_name3,port_name4, ...);

    }

};
```

By position

```
#include<submodule_name.h>

SC_MODULE(submodule_name){
    Module port declarations;

    module_name_A *instance_A;
    module_name_B *instance_B;

    SC_CTOR(module_name){

        instance_A->subport1(port_name1);
        instance_A->subport2(port_name2);
                  ...
        instance_B->subport1(port_name3);
        instance_B->subport2(port_name4);
                  ...
    }

};
```

By name

Process Registration

- The SystemC simulation process is the basic unit of execution
  - A simulation process is a member function of an SC_MODULE that is invoked by the scheduler in the SystemC simulation kernel

- All simulation processes are registered with the SystemC simulation kernel

```
#include<submodule_name.h>

SC_MODULE(submodule_name){
    Module port declarations;

    module_name_A *instance_A;
    module_name_B *instance_B;

    void Proc1();
    void Proc2();
    void Proc3();

    SC_CTOR(module_name){
        (*instance_A)(port_name1,port_name2, ...);
        (*instance_B)(port_name3,port_name4, ...);

        SC_METHOD(Proc1);
        sensitive<< signal1 << signal2 << ...;
        SC_THREAD(Proc2);
        sensitive<< signal3 << signal4 << ...;
        SC_CTHREAD(Proc3,clk.pos());
    }

};
```

Miscellaneous Setup

- In addition to allocating sub-design and registering process, user-defined setup also can define in SC_CTOR
  - Variable initialization
  - Member function that is executed only once

Miscellaneous setup

```
#include<submodule_name.h>

SC_MODULE(submodule_name){
    Module port declarations;

    module_name_A *instance_A;
    module_name_B *instance_B;

    void Proc1();
    void Proc2();
    void Proc3();


    SC_CTOR(module_name){
        (*instance_A)(port_name1,port_name2, ...);
        (*instance_B)(port_name3,port_name4, ...);

        variable1 = 10;
        function();

        SC_METHOD(Proc1);
        sensitive<< signal1 << signal2 << ...;
        SC_THREAD(Proc2);
        sensitive<< signal3 << signal4 << ...;
        SC_CTHREAD(Proc3,clk.pos());
    }
    private:
        void function();
        int variable1;

};
```

Sensitivity

- Processes be resumed or activated by sensitivity
  - Sensitive to events
  - Sensitive to signal edge

- SystemC has two types of sensitivity
  - Static sensitivity
  - Dynamic sensitivity

- Static sensitivity is implemented by applying the SystemC **sensitive**

- Dynamic sensitivity lets a simulation process change its sensitivity on the fly

Timing Annotation

- As a SystemC simulation runs, there are three unique time measurements
  - Wall clock time
  - Process time
  - Simulation time

- Wall clock time
  - The time from the start of execution to completion

- Process time
  - The actual time spent executing the simulation

- Simulation time
  - the time being modeled by the simulation, and it may be less than or greater than the simulation's wall-clock time.

sc_time

- **sc_time** is used to track simulated time and specify delays and timeout
- **sc_time** is represented by a minimum of a 64-bit unsigned integer
- Syntax
  - sc_time name
  - sc_time name(magnitude, timeunits);
- Example
  - sc_time t_period(5, SC_NS);

Enum	Units	Magnitude
SC_FS	femtoseconds	10^{-15}
SC_PS	Picoseconds	10^{-12}
SC_NS	Nanoseconds	10^{-9}
SC_US	Microseconds	10^{-6}
SC_MS	Milliseconds	10^{-3}
SC_SEC	Seconds	10^{0}

Time Units

Time Resolution

- The time resolution is the smallest amount of time that can be represented by all **sc_time** objects in a SystemC simulation
- A user can get the current time resolution by calling the **sc_get_resolution()**
- A user can set the time resolution to other value by calling the **sc_set_time_resolution**
  - The function must be called before any sc_time objects are constructed

sc_time_stamp

- The method **sc_time_stamp**() can be used to obtain the current simulated **time_value**
  - The returned value is an **sc_time** object

- This return value allows **sc_time_stamp**() to be used as any other object for assignment, arithmetic, and comparison operations or to be inserted in an output stream for display

Example

```
cout << "The time is now " << sc_time_stamp() << "!" <, endl;
```

The time is now 0 ns!

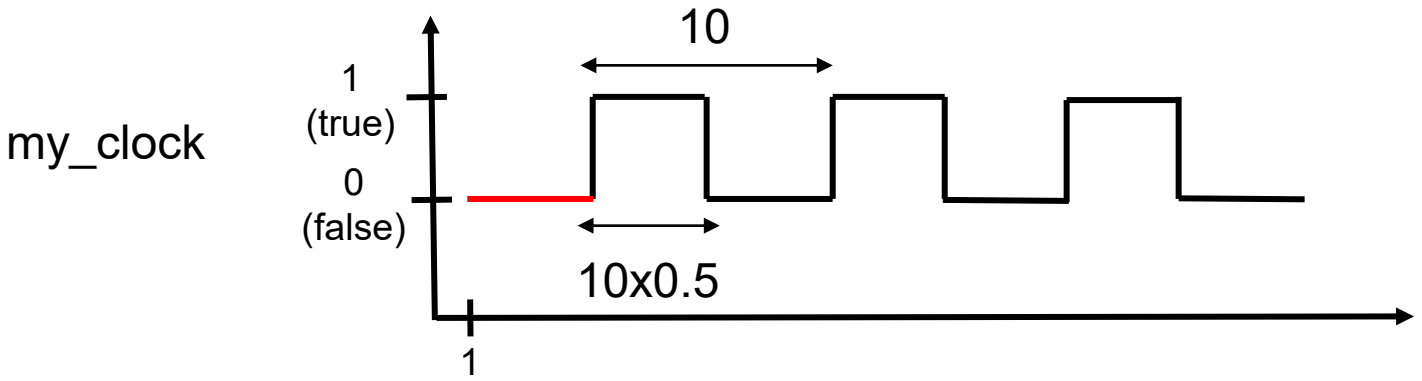
result

sc_clock

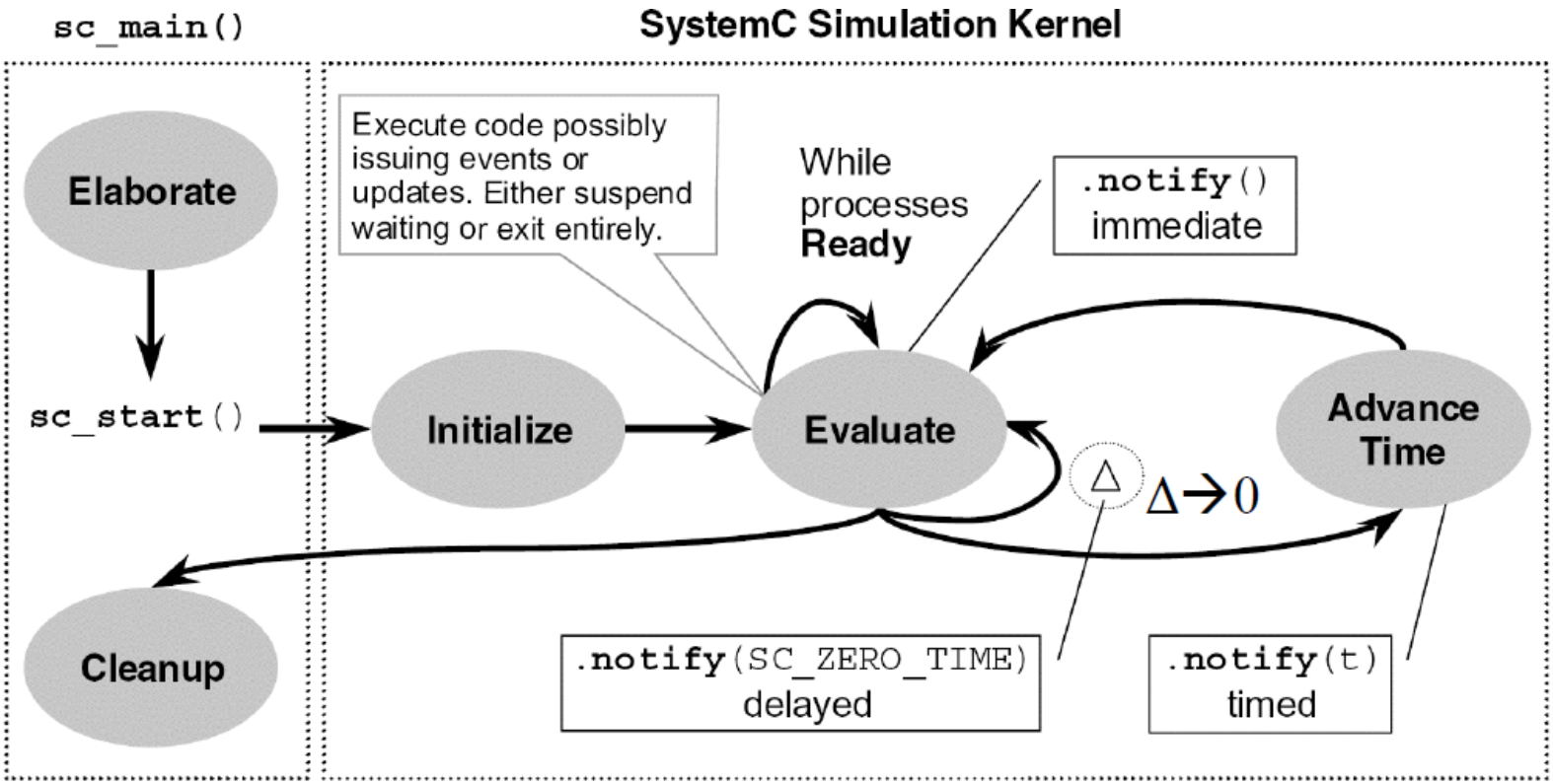
- Clock is special object in SystemC
  - Generate clock
  - Synchronize the events in SystemC model link

- sc_clock(const char* name, double period, double duty_cycle, start_time, bool posedge_first);

- **Example**
  - sc_clock clock("my_clock", 10, 0.5, 1, false);

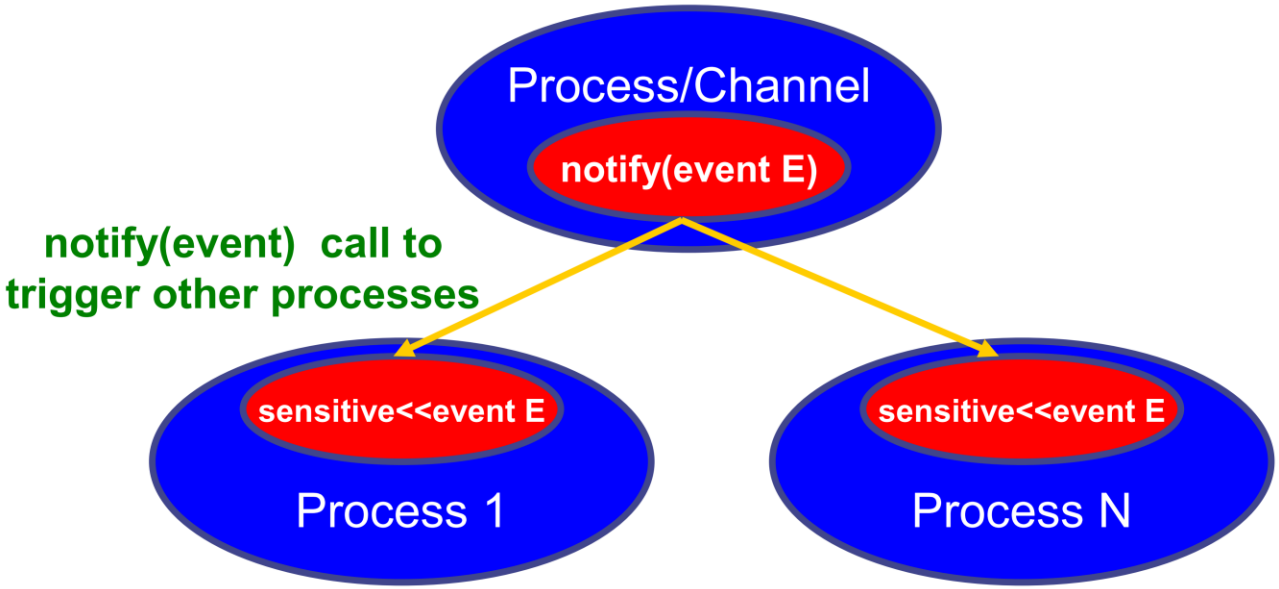


Event-based simulation



Events

- Events can be produced by
  - sc_event
- Events can be sensitive by
  - Change states in process or of channel (static sensitivity)
  - Report change by calling notify() on event (dynamic sensitivity)



Method Process

- Method is just SystemC ways of saying function
  - Method inherit the behavior of function
- Executed when events occur on the sensitivity list
- A locally declared variable is not permanent.
- If execution finish, return control back to simulation kernel

SC_METHOD

- The key word of method process is **SC_METHOD**
- The **SC_METHOD** is in many ways similar to the **SC_THREAD**. However, there have two differences
  - **SC_METHOD** will run more than once by design
  - Methods cannot be suspended by a wait statement during their execution
- Not to write infinite loop within method process
  - Can not call **wait()**

SC_METHOD

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

using namespace std;

SC_MODULE(Hello_SystemC){
    sc_in_clk iclk;

    void method1(){
        cout << sc_time_stamp() << " " <<"Hello world!" << endl;
    }
    SC_CTOR(Hello_SystemC){
        SC_METHOD(method1);
        sensitive << iclk.pos();

    }
};
```

Hello.h

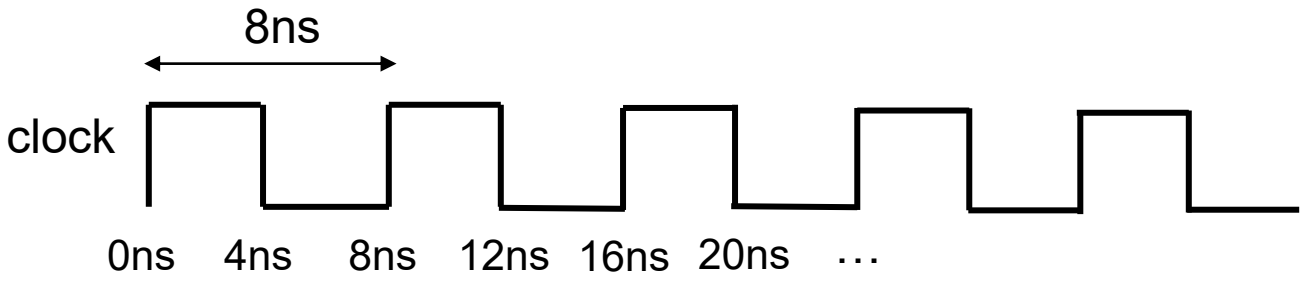
```
int sc_main(int argc, char *argv[])
{
    const sc_time t_PERIOD(8,SC_NS);
    sc_clock clk("clk",t_PERIOD);
    Hello_SystemC iHelloWorld("iHelloWorld");
    iHelloWorld.iclk(clk);
    sc_start(20,SC_NS);
    return 0;

}
```

main.cpp

SC_METHOD

Result

```
0 ns Hello world!
8 ns Hello world!
16 ns Hello world!
```

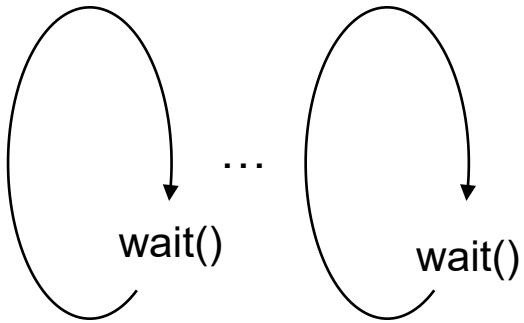


Thread Process

- Thread is a process that always alive, its local variables are alive throughout the simulation.
- Once start to execute, it is in complete control of the simulation
- When terminated, it is gone forever
  - Typically, it contains an infinite loop and at least a *wait*
- A thread can contain **wait()** statements that suspend the process until an event occurs on one of the signals the process is sensitive to.

SC_THREAD

- Key word: **SC_THREAD**
- Can be suspended and reactivated by **wait()** call
- Usually contain infinite loop, process continue to execute until the next wait()
  - wait() : Wait for the sensitive list event to occur.
  - wait(int) : Wait for n events to occur, events are the one in sensitive list.
  - wait(double, sc_time_unit) : Wait for specified time.



Threads in SystemC

wait(sc_time)

- Example

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

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

SC_THREAD

```
SC_MODULE(process){
    sc_in<int> x_in1,x_in2;
    sc_out<int> x_out;
    void run();

    SC_CTOR(process){
        SC_THREAD(run);
        sensitive << x_in1,x_in2;
    }
};
```

process.h

```
void process::run(){
    while(true){    //infinite loop
        if(x_in1 != 0)
            x_out = x_in1;
        else
            x_out = x_in2;
        wait();     // use wait() call to suspend process
                    // until xin signal change
    }
}
```

process.cpp

SC_THREAD

```
SC_MODULE(pattern_gen)
{
    sc_in_clk clock;
    sc_out<int> x_in1,x_in2;


    void pattern1()
    {
        x_in1 = rand();
        x_in2 = rand();
    }

    SC_CTOR(pattern_gen)
    {

        SC_METHOD(pattern1);
        sensitive << clock.pos();
    }
};
```

Create pattern

```
#include "patterm_gen.h"
#include "test.h"
#include <iostream>
using namespace std;

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

    sc_signal<int> x_in1,x_in2,x_out;
    sc_clock clock("My_CLOCK",10,5,0,1);

    pattern_gen pattern("Generation");
    process proc("process");

    pattern(clock,x_in1,x_in2);
    proc(x_in1,x_in2,x_out);


    sc_start(50,SC_NS);


    return 0;
}
```

Main.cpp

sc_clock clock(“My_CLOCK”, 10, 0.5, 0, 1)

```
0 ns 0
10 ns 1714636915
20 ns 424238335
30 ns 1649760492
40 ns 1189641421
50 ns 1350490027
```

Result

SC_THREAD

```
void process::run(){
    while(true){    //infinite loop
        if(x_in1 !=0)
            x_out = x_in1;
        else
            x_out = x_in2;

        wait(3,SC_NS);      //use wait() call to suspend process
                            // until  sin signal change
    }
}
```

```
0 ns 0
3 ns 1714636915
6 ns 1714636915
9 ns 1714636915
12 ns 424238335
15 ns 424238335
18 ns 424238335
21 ns 1649760492
24 ns 1649760492
27 ns 1649760492
30 ns 1649760492
33 ns 1189641421
36 ns 1189641421
39 ns 1189641421
42 ns 1350490027
45 ns 1350490027
48 ns 1350490027
```

Clocked Thread Process

- It is a special case of a thread process.
  - Clocked Thread Processes are only triggered on one edge of one clock.
- Key word: **SC_CTHREAD**
- **SC_CTHREAD** is not recommended
  - It can be fully replaced by a normal thread
- wait_until() is a function can be used in clock thread.
  - It halt the execution of the process until a specific event has occurred

SC_CTHREAD

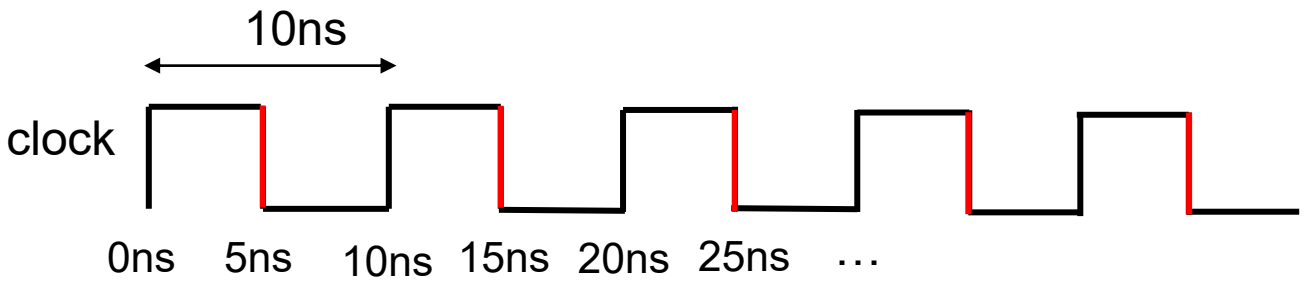
```
#include "systemc.h"

SC_MODULE(process){
    sc_in_clk clk;
    sc_in<int>x_in1,x_in2;
    sc_out<int>x_out;

    void run();

    SC_CTOR(process){
        SC_CTHREAD(run,clk.neg());
    }

};
```

```
void process:run(){
    while(true){
        x_out = x_in1 + x_in2;
        wait();
        cout<< sc_time_stamp() <<" "<< x_out <<endl;
    }
}
```

SC_CTHREAD

Result

5 ns 0
15 ns -898637604
25 ns -1912981168
35 ns -1925321418
45 ns 1786158070



SC_THREAD and SC_CTHREAD

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

SC_THREAD and SC_CTHREAD

- A thread process instance may have static sensitivity or may call function wait to create dynamic sensitivity. A clocked thread process instance is statically sensitive only to a single clock.

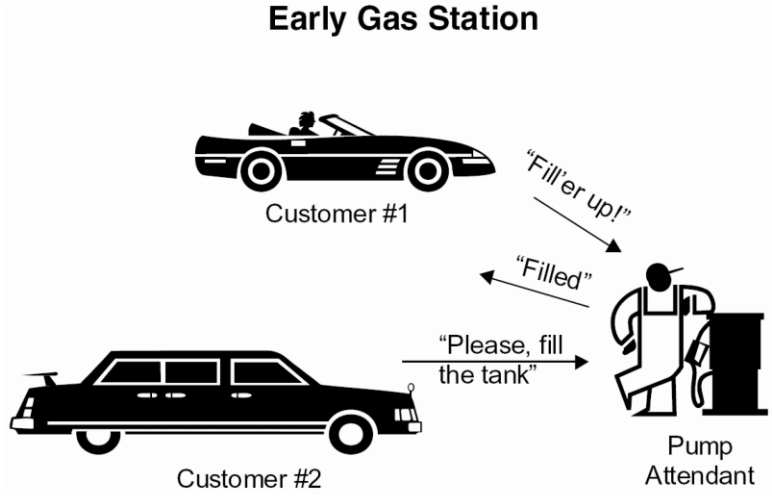
- Each thread process requires its own execution stack. As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.

- If the thread or clocked thread process executes the entire function body or executes a return statement and thus returns control to the kernel, the associated function shall not be called again for that process instance. The process instance is then said to be terminated.

Comparison on Processeswqw1aesghfkljo

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

Example – Gas Station




```
SC_MODULE(gas_station) {
  sc_event e_request1, e_request2;
  sc_event e_tank_filled;
  SC_CTOR(gas_station) {
    SC_THREAD(customer1_thread);
      sensitive(e_tank_filled); // functional
                                // notation
    SC_METHOD(attendant_method);
      sensitive << e_request1
                << e_request2; // streaming notation
    SC_THREAD(customer2_thread);
  }
  void attendant_method();
  void customer1_thread();
  void customer2_thread();
};
```

Implementation

```
…
void gas_station::customer1_thread() {
  while (true) {
    wait(EMPTY_TIME);
    cout << "Customer1 needs gas" << endl;
    m_tank1 = 0;
    do {
      e_request1.notify();
      wait(); // use static sensitivity
    } while (m_tank1 == 0);
  }//endforever
}//end customer1_thread()

// omitting customer2_thread (almost identical
// except using wait(e_request2);)
```

```
void gas_station::attendant_method() {
  if (!m_filling) {
      …
      cout << "Filling tank" << endl;
      m_filling = true;
      next_trigger(FILL_TIME);
      …
  } else {
      …
      e_filled.notify(SC_ZERO_TIME);
      cout << "Filled tank" << endl;
    …
    m_filling = false;
    …
  }//endif
}//end attendant_method()
```

Dynamic Sensitivity

- Sensitive by calling function as process execution
- SC_METHOD : next_trigger()
- SC_THREAD : wait()
- SC_CTHREAD : wait_until(), watching() (Not support in SystemC 2.2)

Dynamic Sensitivity-next_trigger()

- Used on SC_METHOD
  - Process invoked only when next_trigger() event occurred

```
#include "systemc.h"

SC_MODULE(process){
    sc_in<int>x_in1,x_in2;
    sc_out<int>x_out;

    void run();

    SC_CTOR(process){
        SC_METHOD(run);
        sensitive << x_in1 << x_in2;
    }

};
```

process.h

```
void process::run(){
    if(x_in1 !=0)
        x_out = x_in1;
    else
        x_out = x_in2;

    next_trigger(3,SC_NS);
    cout<< sc_time_stamp() <<" "<< x_out <<endl;

}
```

process.cpp

Dynamic Sensitivity-next_trigger()

- Show the result every 3ns

result

```
0 ns 0
3 ns 1714636915
6 ns 1714636915
9 ns 1714636915
12 ns 424238335
15 ns 424238335
18 ns 424238335
21 ns 1649760492
24 ns 1649760492
27 ns 1649760492
30 ns 1649760492
33 ns 1189641421
36 ns 1189641421
39 ns 1189641421
42 ns 1350490027
45 ns 1350490027
48 ns 1350490027
```

dont_initialize

- The simulation engine description specifies that processes are executed at least once initially by placing processes
- It may be necessary to specify that some processes should not be made runnable at initialization
  - SystemC provides the **dont_initialize**() method

Hello.h

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

using namespace std;

SC_MODULE(Hello_SystemC){
    sc_in_clk iclk;

    void method1(){
        cout << sc_time_stamp() << " " <<"Hello world!" << endl;
    }
    SC_CTOR(Hello_SystemC){
        SC_METHOD(method1);
        sensitive << iclk.pos();
        dont_initialize();
    }
};
```

dont_initialize

- Process and threads (not the cthread) get executed automatically in constructor even if event in sensitivity list does not occur.
  - Execute once more

Without don't_initialize

With don't_initialize

Parallel simulation

- Consider a design with four thread processes

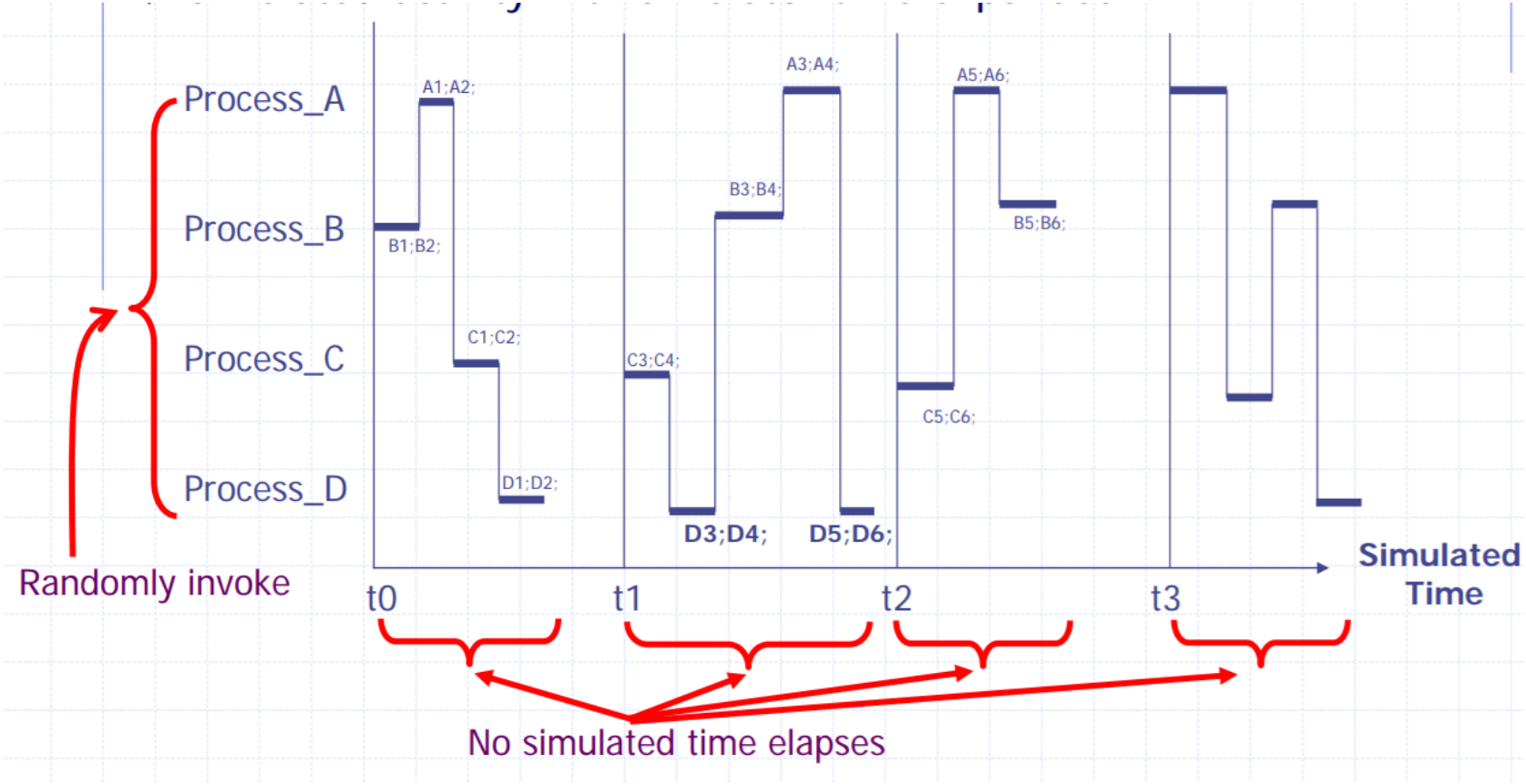
```
Process_A(){
   stmt_A1;
   stmt_A2;
   wait(t1);
   stmt_A3;
   stmt_A4;
   wait(t2);
   stmt_A5;
   stmt_A6;
   wait(t3);
}
```

```
Process_B(){
   stmt_B1;
   stmt_B2;
   wait(t1);
   stmt_B3;
   stmt_B4;
   wait(t2);
   stmt_B5;
   stmt_B6;
   wait(t3);
}
```

```
Process_C(){
   stmt_C1;
   stmt_C2;
   wait(t1);
   stmt_C3;
   stmt_C4;
   wait(t2);
   stmt_C5;
   stmt_C6;
   wait(t3);
}
```

```
Process_D(){
   stmt_D1;
   stmt_D2;
   wait(t1);
   stmt_D3;
   stmt_D4;
   wait(t2);
   stmt_D5;
   stmt_D6;
   wait(t3);
}
```

Parallel simulation



Parallel simulation

```
#include "systemc.h"

SC_MODULE(process){
    sc_in<bool>en;

    void process_A();
    void process_B();
    void process_C();
    void process_D();

    SC_CTOR(process){
        SC_THREAD(process_A);
        SC_THREAD(process_B);
        SC_THREAD(process_C);
        SC_THREAD(process_D);
        sensitive << en;;
    }

};
```

```
void process::process_A(){
    int a;
    a = 10;
    cout << sc_time_stamp() << " a= " << a << endl;
    wait(2,SC_NS);
    a = 20;
    cout << sc_time_stamp() << " a= " << a << endl;
    wait(3,SC_NS);
    a = 30;
    cout << sc_time_stamp() << " a= " << a << endl;
    wait(4,SC_NS);
}


void process::process_B(){
    int b;
    b = 10;
    cout << sc_time_stamp() << " b= " << b << endl;
    wait(2,SC_NS);
    b = 20;
    cout << sc_time_stamp() << " b= " << b << endl;
    wait(3,SC_NS);
    b = 30;
    cout << sc_time_stamp() << " b= " << b << endl;
    wait(4,SC_NS);
}
```

Parallel simulation

```
void process::process_C(){
    int c;
    c = 10;
    cout << sc_time_stamp() << " c= " << c << endl;
    wait(2,SC_NS);
    c = 20;
    cout << sc_time_stamp() << " c= " << c << endl;
    wait(3,SC_NS);
    c = 30;
    cout << sc_time_stamp() << " c= " << c << endl;
    wait(4,SC_NS);
}


void process::process_D(){
    int d;
    d = 10;
    cout << sc_time_stamp() << " d= " << d << endl;
    wait(2,SC_NS);
    d = 20;
    cout << sc_time_stamp() << " d= " << d << endl;
    wait(3,SC_NS);
    d = 30;
    cout << sc_time_stamp() << " d= " << d << endl;
    wait(4,SC_NS);
}
```

0 ns a = 10
0 ns b =10
0 ns c =10
0 ns d =10
2 ns a =20
2 ns d =20
2 ns c =20
2 ns b =20
5 ns b =30
5 ns c =30
5 ns d =30
5 ns a = 30

Structure hierarchy

- There are four approaches of structure hierarchy
  - Direct top-level (**sc_main**)
  - Indirect top-level (**sc_main**)
  - Direct submodule
  - Indirect submodule
- Design hierarchy in SystemC uses instantiations of modules as member data of parent modules.

Direct Top-Level Implementation

- The hierarchy instances are written in the **sc_main**

- The name given via the constructor is the hierarchical instance name used by the SystemC kernel and is very useful during debug.

```
//FILE: main.cpp
#include <systemc>
#include "Car.h"
int sc_main(int argc, char* argv[]) {
    Car car_i("car_i");
    sc_start();
    return 0;
}
```

Indirect Top-Level Implementation

- This approach adds two lines of syntax with both a pointer declaration and an instance creation via new.
- It adds the possibility of dynamically configuring the design with the addition of **if-else** and looping constructs.

```
//FILE: main.cpp
#include <systemc>
#include "Car.h"
int sc_main(int argc, char* argv[]) {
    Car* car_iptr; // pointer to Car
    car_iptr = new Car("car_i"); // create Car
    sc_start();
    delete car_iptr;
    return 0;
}
```

Direct Submodule Implementation

- **SC_CTOR** is a macro that hides the C++ constructor syntax.
- The constructor implemented in **SC_CTOR** requires initialization with a SystemC instance name, therefore the requirement to initialize the submodules

```
//FILE:Car.h
#include "Body.h"
#include "Engine.h"
SC_MODULE(Car) {
    Body body_i;
    Engine eng_i ;
    SC_CTOR(Car)
    : body_i("body_i") //initialization
    , eng_i("eng_i") //initialization
    {
    // other initialization
    }
};
```

Indirect Submodule Implementation

- Use of indirection renders the instantiation a little bit easier to read for the submodule header-only case.
- However, no other advantages are clear.

```
//FILE:Body.h
#include "Wheel.h"
SC_MODULE(Body) {
    Wheel* wheel_FL_iptr;
    Wheel* wheel_FR_iptr;
    SC_CTOR(Body) {
    wheel_FL_iptr = new Wheel("wheel_FL_i");
    wheel_FR_iptr = new Wheel("wheel_FR_i");
    // other initialization
    }
};
```

Contrasting Implementation Approaches

Level	Allocation	Pros	Cons
Main	Direct	Least code	Inconsistent with other levels
Main	Indirect	Dynamically configurable	Involves pointers
Module	Direct header only	All in one file Easier to understand	Requires submodule headers
Module	Indirect header only	1. All in one file 2. Dynamically configurable	1. Involves pointers 2. Requires submodule headers

Summary

- A module is the basic functional block in SystemC
- SystemC provide three type of process to simulate the hardware behavior
- Module hierarchy is a very important concept to design hardware