SISTEME DE CALCUL DEDICATE

Curs 4

Outline

- SystemC
  - Dynamic processes
  - Basic channels
  - Evaluate-update channels
- Bibliography

Dynamic processes

- SystemC 2.1 introduced the concept of ***dynamically spawned processes***
- useful in testbench scenarios
  - to track transaction completion
  - to spawn traffic generators dynamically

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
```

Dynamic processes

- declare the functions to be spawned as processes
- unlike static processes
  - dynamic processes may have up to eight arguments and a return value
    - the return value will be provided via a reference variable in the actual spawn function

```
// Ordinary function declarations
void inject(void); // no args or return
int count_changes(sc_signal<int>& sig);

// Method function declarations
class TestChan : public sc_module {
  …
  bool Track(sc_signal<packet>& pkt);
  void Errors(int maxwarn, int maxerr);
  void Speed(void);
  …
};
```

Dynamic processes

- define the implementation and register the function with the kernel
  - within an **SC_THREAD**
  - with restrictions within an **SC_METHOD**
- syntax to register dynamic processes with void return

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    sc_bind(&funcName, ARGS…)//no return value
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    sc_bind(&methName, object, ARGS…)//no return
   ,processName
   ,spawnOptions
  );
```

Dynamic processes

- syntax to register dynamic processes with return values
- object is a reference to the calling module
  - normally just use the C++ keyword **this**

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    &returnVar
   ,sc_bind(&funcName, ARGS…)
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    &returnVar
   ,sc_bind(&methodName, object, ARGS …)
   ,processName
   ,spawnOptions
  );
```

Dynamic processes

- by default, arguments are passed by value
- to pass by reference or by constant reference, a special syntax is required

```
sc_ref(var)   // reference
sc_cref(var)  // constant reference
```

Dynamic processes

- spawn options are determined
  - by creating an **sc_spawn_option** object
  - invoking one of several methods that set the options

```
sc_spawn_option objname;
objname.spawn_method();// register as SC_METHOD
objname.dont_initialize();
objname.set_sensitivity(event_ptr);
objname.set_sensitivity(port_ptr);
objname.set_sensitivity(interface_ptr);
objname.set_sensitivity(event_finder_ptr);
objname.set_stack_size(value); // experts only!
```

Dynamic processes

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
…
void spawned_thread() {// This will be spawned
  cout << "INFO: spawned_thread "
       << sc_get_current_process_handle().name()
       << " @ " << sc_time_stamp() << endl;
  wait(10,SC_NS);
  cout << "INFO: Exiting" << endl;
}

void simple_spawn::main_thread() {
  wait(15,SC_NS);
  // Unused handle discarded
  sc_spawn(sc_bind(&spawned_thread));
  cout << "INFO: main_thread " << name()
       << " @ " << sc_time_stamp() << endl;
  wait(15,SC_NS);
  cout << "INFO: main_thread stopping "
       << " @ " << sc_time_stamp() << endl;
}
```

Dynamic processes

```
// Add "& resume" to sensitivity while suspended
void sc_process_handle::suspend(descend);
void sc_process_handle::resume(descend);

// Ignore sensitivity while disabled
void sc_process_handle::disable(descend);
void sc_process_handle::enable(descend);

// Complete remove process
void sc_process_handle::kill(descend);

// Asynchronously restart a process
void sc_process_handle::reset(descend);

// Reset process on every resumption event
void sc_process_handle::sync_reset_on(descend);
void sc_process_handle::sync_reset_off(descend);

// Throw an exception in the specified process
template<typename T>
  void sc_process_handle::throw_it(
                                const T&,descend);
```

Basic channels

- communication of information between concurrent processes is done using
  - events
    - require careful coding
    - use handshake variable
  - ordinary module member data
  - channels - encapsulate complex communications
    - primitive
    - hierarchical

Basic channels

- primitive channels
  - inherit from the base class **sc_prim_channel**
  - also inherit from and implement one or more SystemC interface classes
  - types
    - **sc_mutex**
    - **sc_semaphore**
    - **sc_fifo**<*T*>

Basic channels

- mutex is short for ***mutually exclusive text***
- **sc_mutex** class
  - implements **sc_mutex_if** interface class
- blocking methods can only be used in **SC_THREAD** processes

```
sc_mutex NAME;

NAME.lock();       // Lock the mutex,
                   // wait until unlocked if in use
int NAME.trylock() // Non-blocking, returns success

NAME.unlock();     // Free a previously locked mutex
```

Basic channels

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

```
void grab_bus_method() {
  if (bus_access.trylock() == 0) {
    // access bus
    …
    bus_access.unlock();
  }
}
```

Basic channels

- **sc_semaphore** class
  - inherits from and implements the **sc_semaphore_if** class

```
sc_semaphore NAME(COUNT);

NAME.wait();            // Lock one semaphore
                        // Wait until available if in use
int NAME.trywait()   // Non-blocking, return success

int NAME.get_value() // Returns available semaphores

NAME.post();            // Free one previously locked
                        // semaphore
```

Basic channels

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  …
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, const int& data) {
    write_ports.wait();
    // perform write
    write_ports.post();
  }
  …
};//endclass
```

Basic channels

- the most popular channel for modeling at the architectural level is the **sc_fifo**<*T*> channel
  - inherits from and implements two interface classes:
    - **sc_fifo_in_if**<*T*>
    - **sc_fifo_out_if**<*T*>

Basic channels

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);

NAME.write(VALUE);
NAME.read(REFERENCE);
… = NAME.read() /* function style */
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());
```

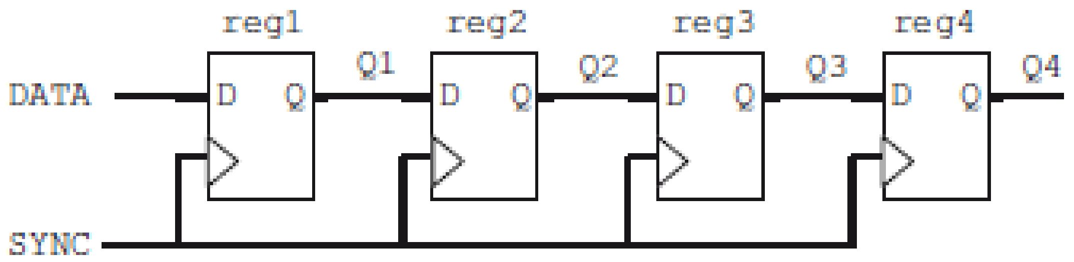
Basic channels

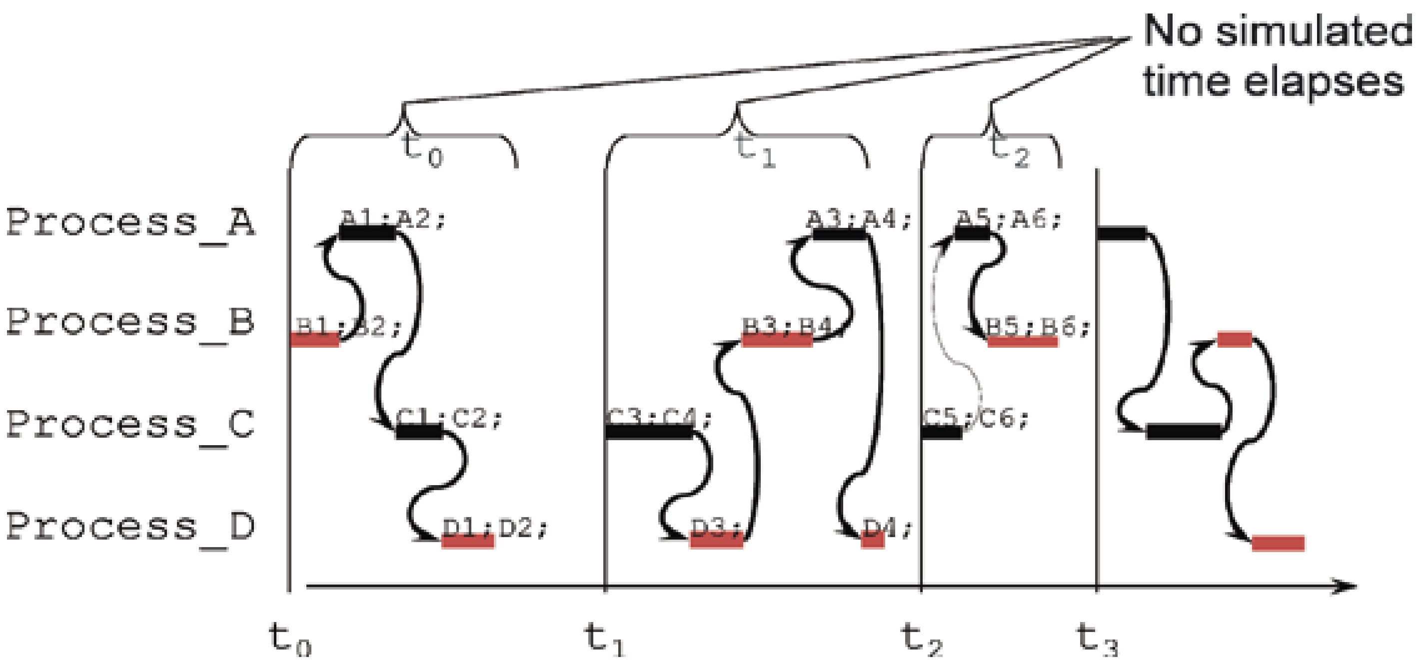
```
SC_MODULE(kahn_ex) {
  …
  sc_fifo<double> a, b, y;
  …
};
// Constructor
kahn_ex::kahn_ex() : a(24), b(24), y(48)
{
  …
}
void kahn_ex ::stim_thread() {
  for (int i=0; i!=1024; ++i) {
    a.write(double(rand()/1000));
    b.write(double(rand()/1000));
  }
}
void kahn_ex::addsub_thread() {
  while(true) {
    y.write(kA*a.read() + kB*b.read());
    y.write(kA*a.read() - kB*b.read());
  }//endforever
}
void kahn_ex::monitor_method() {
  cout << y.read() << endl;
}
```

Evaluate-Update Channels

- electronic hardware
  - behave in a manner approaching instantaneous activity
  - electronic signals have
    - a single source (producer)
    - multiple sinks (consumer)
      - it is quite important that all sinks “see” a signal update at the same time

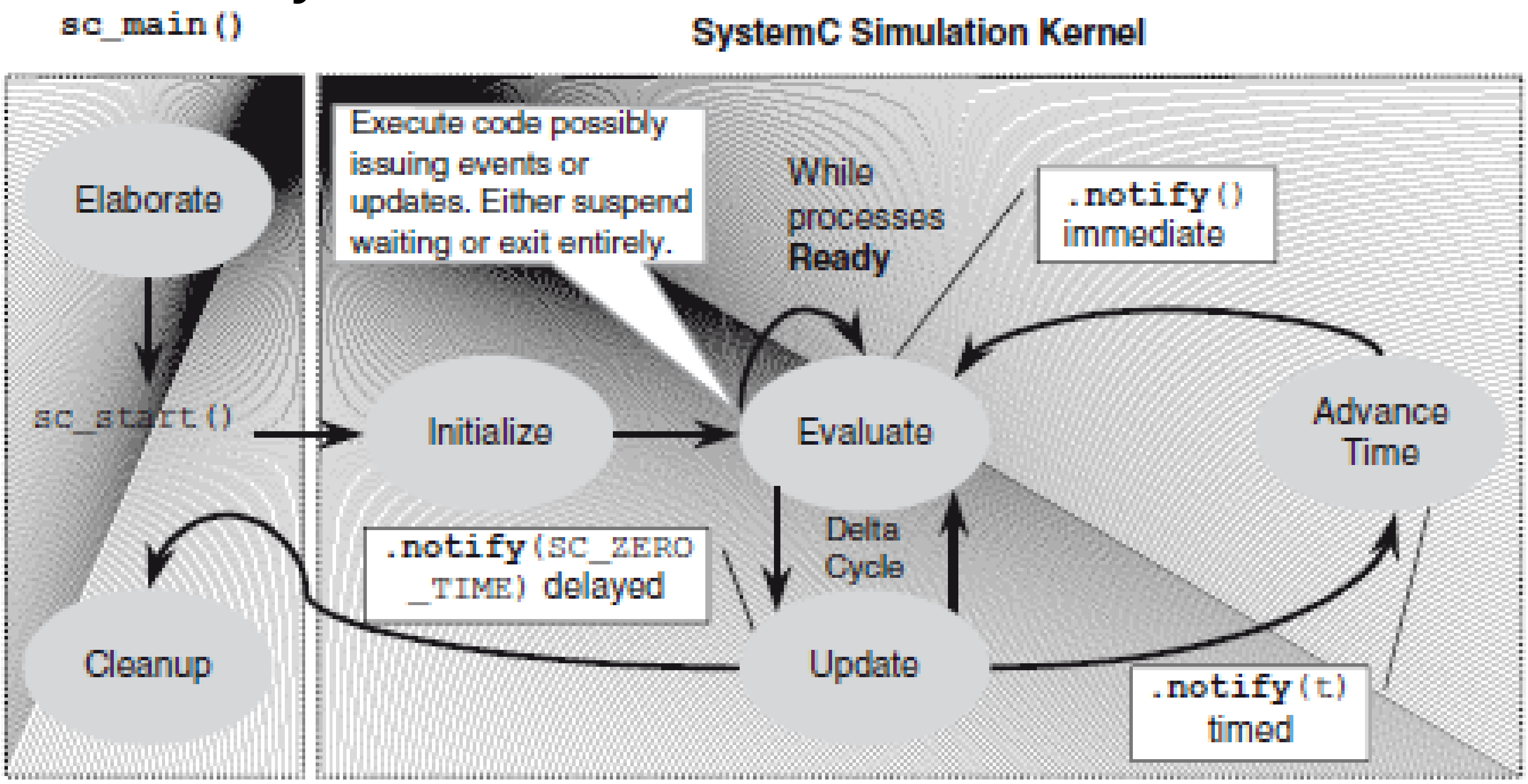
Evaluate-Update Channels





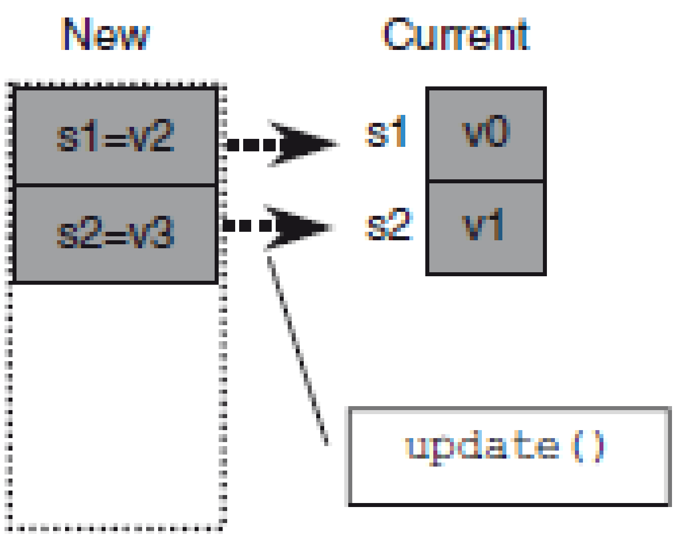
Evaluate-Update Channels

- evaluate-update paradigm
  - delta-cycle



Evaluate-Update Channels

- signal channels, use update phase as a point of data synchronization
- to accomplish this synchronization, every channel has two storage locations:
  - the current value and the new value



Evaluate-Update Channels

- **sc_signal**<*T*> primitive channel and its close relative, **sc_buffer**<*T*> both use the evaluate-update paradigm

```
  sc_signal<datatype> signame[, signame_i]…;//define
…
  signame.write(newvalue);
  varname  = signame.read();
  wait(signame.value_changed_event()|...);
  wait(signame.default_event()|...);
  if (signame.event() == true) {
    // occurred in previous delta-cycle
```

```
// Declare variables
int                  count;
string               message_temp;
sc_signal<int>       count_sig;
sc_signal<string>    message_sig;

cout << "Initialize during 1st delta cycle" << endl;
count_sig.write(10);
message_sig.write("Hello");
count = 11;
message_temp = "Whoa";
cout << "count is " << count << " "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "' "
     << "message_sig is '" << message_sig << "'"
     << endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "2nd delta cycle" << endl;
count = 20;
count_sig.write(count);
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     <<endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "3rd delta cycle" << endl;
message_sig.write(message_temp = "Rev engines");
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     << endl << endl << "Done" << endl;
```

```
Initialize during 1st delta cycle
count is 11, count_sig is 0
message_temp is 'Whoa', message_sig is ''
Waiting

2nd delta cycle
count is 20, count_sig is 10
message_temp is 'Whoa', message_sig is 'Hello'
Waiting

3rd delta cycle
count is 20, count_sig is 20
message_temp is 'Rev engines', message_sig is
'Hello'

Done
```

Evaluate-Update Channels

- multiple writers

```
sc_signal_resolved name;
sc_signal_rv<WIDTH> name;
```

Multiple Drivers on a Bus

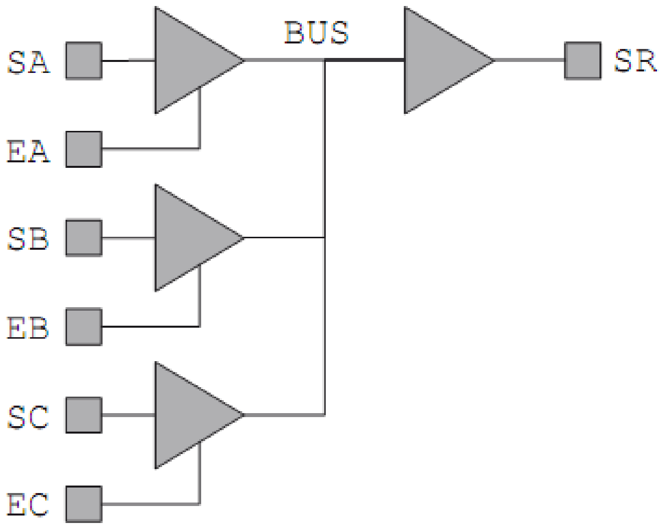


Table 9.1 Resolution functionality for `sc_signal_resolved`

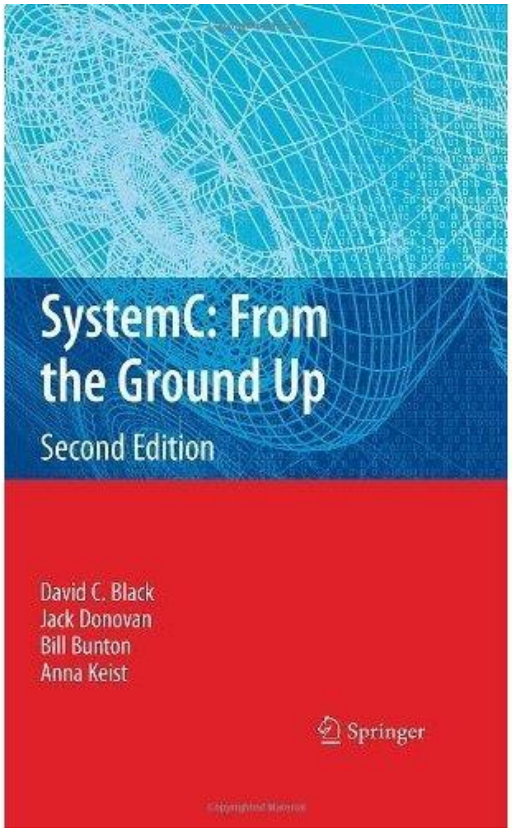
A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

Evaluate-Update Channels

- template specializations
  - a template specialization occurs when a definition is provided for a specific template value
  - specialized templates
    - sc_signal<bool>
    - sc_signal<sc_logic>

```
sensitive << signame.posedge_event()
          << signame.negedge_event();
wait(signame.posedge_event()
    |signame.negedge_event());
if (signame.posedge_event()
   |signame.negedge_event()) {
```

Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”