# ECE382M.20:
# System-on-Chip (SoC) Design

## Lecture 3 – The SystemC Language

***Sources:***
***M. Radetzki, Univ. of Stuttgart***

Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu

The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
*Cockrell School of Engineering*

## Lecture 3: Outline

- **System design languages**
  - Goals, requirements
  - Separation of computation & communication
- **The SystemC language**
  - Basic SystemC model structure
  - Core language syntax
  - Data types
  - Events and simulation semantics
  - Channel library
  - Example

 © 2023 A. Gerstlauer 

(c) 2023 A. Gerstlauer 

## Languages

- **Represent a model in machine-readable form**
  - Execute (simulate)
  - Apply synthesis and verification algorithms and tools
- **Models vs. languages**
  - Languages capture models
  - Model: conceptual notion, Language: concrete form
  - Different languages can capture the same model
  - One language can capture different models
- **Semantics vs. syntax**
  - Syntax defines grammar (textual or graphical)
  - Semantics defines meaning (operational or denotational)
  - Underlying execution/simulation model



## Design Languages

- **Netlists**
  - Structure only: components and connectivity
  - ➢ Gate-level [EDIF], system-level [SPIRIT/XML]
- **Hardware description languages (HDLs)**
  - Event-driven behavior: signals/wires, clocks
  - Register-transfer level (RTL): boolean logic
  - ➢ Discrete event [VHDL, Verilog]
- **System-level design languages (SLDLs)**
  - Software behavior: sequential functionality/programs
  - ➢ C-based, event-driven [SpecC, SystemC, SystemVerilog]

➢ **Structural descriptions at varying levels**
  - ➢ Netlist/architecture + component behavior
  - ➢ Concurrency & logical time

# System-Level Design Languages (SLDLs)

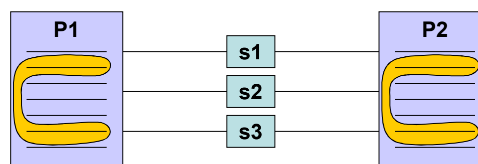
- **C/C++**
  - ANSI standard programming languages, software design
  - Traditionally used for system design because of practicality, availability
- **SystemC**
  - C++ API and class library
  - Initially developed at UC Irvine, IEEE standard by Open SystemC Initiative (OSCI)
- **SpecC**
  - C extension
  - Developed at UC Irvine, standard by SpecC Technology Open Consortium (STOC)
- **SystemVerilog**
  - Verilog with C extensions for testbench development
- **Matlab/Simulink**
  - Specification and simulation in engineering, algorithm design
- **Unified Modeling Language (UML)**
  - Software specification, graphical, extensible (meta-modeling)
  - Modeling and Analysis of Real-time and Embedded systems (MARTE) profile
- **IP-XACT**
  - XML schema for IP component documentation, standard by SPIRIT consortium
- **Rosetta (formerly SLDL)**
  - Formal specification of constraints, requirements
- **SDL**
  - Telecommunication area, standard by ITU
- **…**

*Source: R. Doemer, UC Irvine*



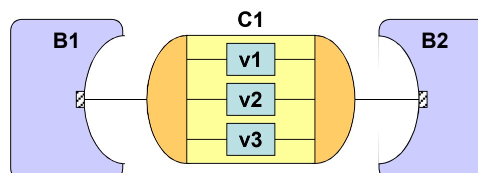
# Computation vs. Communication

- **Traditional model**




  - Processes and signals
  - Mixture of computation and communication
  - Automatic replacement impossible

- **SLDL model**

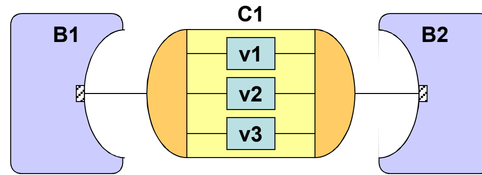



  - Behaviors and channels
  - Separation of computation and communication
  - Plug-and-play

*Source: SpecC Language, R. Doemer, UC Irvine*

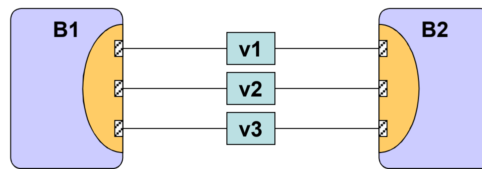
## Computation vs. Communication

- **Protocol Inlining**
  - Specification model
  - Exploration model




    - Computation in behaviors
    - Communication in channels

  - Implementation model




    - Channel disappears
    - Communication inlined into behaviors
    - Wires exposed

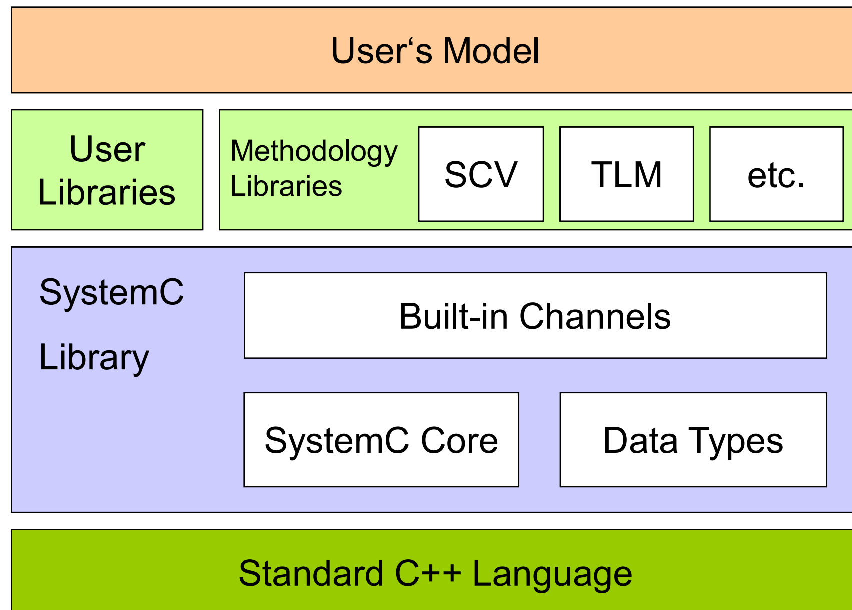
*Source: SpecC Language, R. Doemer, UC Irvine*



## Lecture 3: Outline

- ✓ **System design languages**
  - ✓ Goals, requirements
  - ✓ Separation of computation & communication
- **The SystemC language**
  - Basic SystemC model structure
  - Core language syntax
  - Data types
  - Events and simulation semantics
  - Channel library
  - Example

## SystemC Class Library Structure

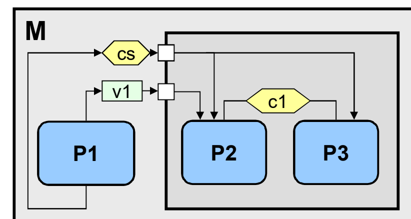



*Source: M. Radetzki, Univ. of Stuttgart*

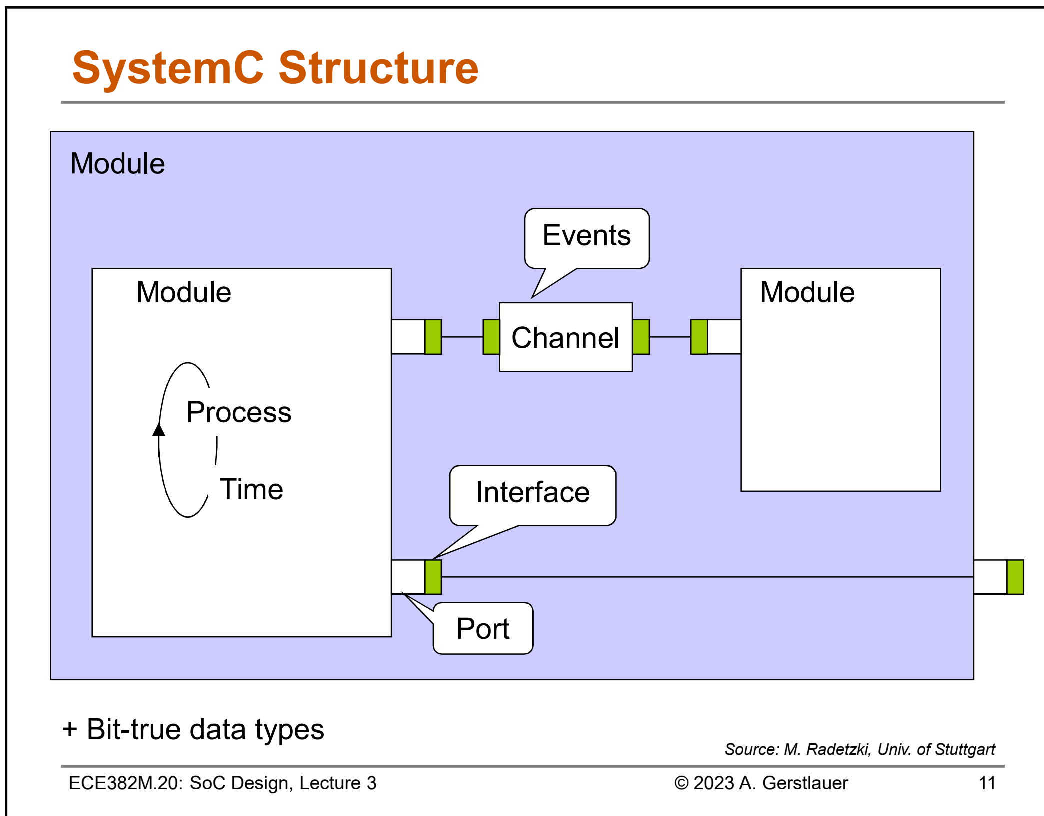


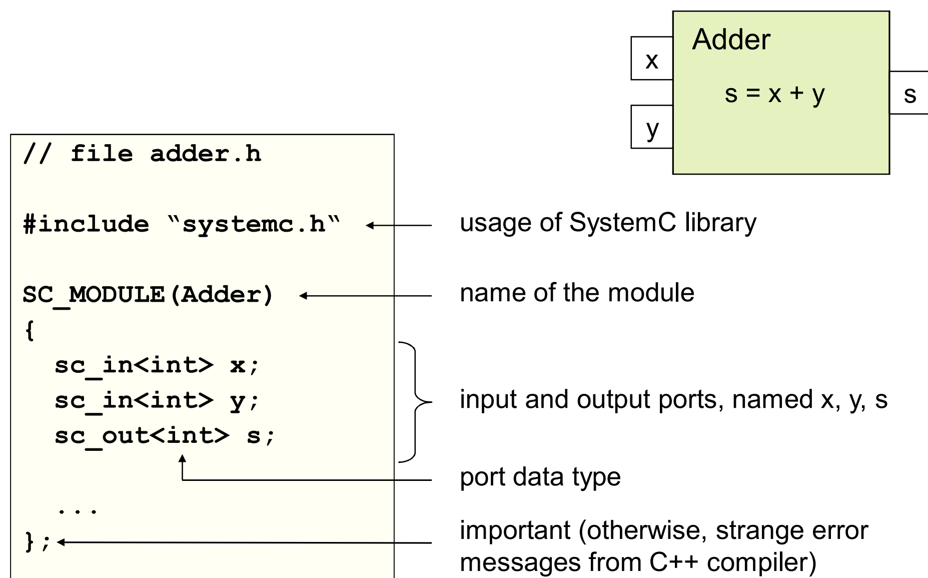
## The SystemC Language

- **SystemC structural hierarchy**
  - Modules
  - Ports and variables
  - Channels* and interfaces*
- **SystemC behavioral hierarchy**
  - Parallel leaf processes
    - SC_METHOD (combinational)
    - SC_THREAD (behavior)




** SystemC 2.0*

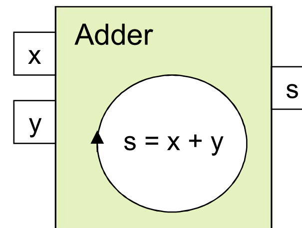

SystemC Structure
Module
Events
Module
Channel
Module
Process
Time
Interface
Port
+ Bit-true data types
Source: M. Radetzki, Univ. of Stuttgart
ECE382M.20: SoC Design, Lecture 3
© 2023 A. Gerstlauer
11


Modules and Ports
Adder
x
y
s = x + y
s
// file adder.h
#include "systemc.h"
SC_MODULE(Adder)
{
sc_in<int> x;
sc_in<int> y;
sc_out<int> s;
...
};
usage of SystemC library
name of the module
input and output ports, named x, y, s
port data type
important (otherwise, strange error messages from C++ compiler)
Source: M. Radetzki
ECE382M.20: SoC Design, Lecture 3
© 2023 A. Gerstlauer
12

## SC_METHOD Process Declaration

```
// file adder.h
#include "systemc.h"
SC_MODULE(Adder)
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive << x << y;
  }
};
```




- `void add();` — function prototype
- `SC_CTOR(Adder)` — module constructor
- `SC_METHOD(add);` — function registered as a process
- `sensitive << x << y;` — activation condition of the process: new value (value change) on port x or port y leads to automatic start of add()

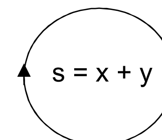
*Source: M. Radetzki*



## SC_METHOD Implementation

```
// file adder.h
#include "systemc.h"
SC_MODULE(Adder)
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_METHOD(add);
    sensitive << x << y;
  }
};
```




```
// file adder.cpp
#include "adder.h"
void Adder::add()
{
  s = x + y;
}
```

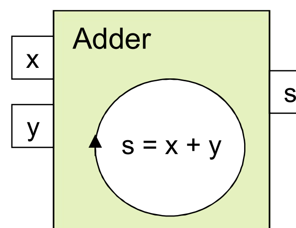
**Alternative:**
`s.write(x.read()+y.read());`

*Source: M. Radetzki*

## SC_THREAD Process Declaration

```
// file adder.h
#include "systemc.h"
SC_MODULE(Adder)
{
  sc_in<int> x;
  sc_in<int> y;
  sc_out<int> s;

  void add();
  SC_CTOR(Adder)
  {
    SC_THREAD(add);
    sensitive << x << y;
  }
};
```




function prototype

module constructor

function registered as a process

activation condition defined, but **no automatic start** of SC_THREAD

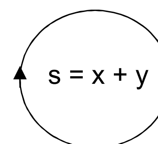
*Source: M. Radetzki*



## SC_THREAD Implementation

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

void Adder::add()
{
  for(;;) // infinite loop
  {
    wait();
    s = x + y;
  }
}
```




SC_THREAD is started only once, at the beginning of the simulation

SC_THREAD specifies activation by call to wait function; here: waits for `sensitive` condition; in adder.h:

```
sensitive << a << b;
```

**The above SC_THREAD implementation has the same functionality as the previous SC_METHOD implementation.**

*Source: M. Radetzki*

# SC_METHOD vs. SC_THREAD

### SC_METHOD

- Started again whenever activation condition triggers
- Must not call **`wait()`**
- Must not block
- Must not contain infinite loop (would block all other processes)
- May use non-blocking communications only
- Must not call functions that block or call **`wait()`**

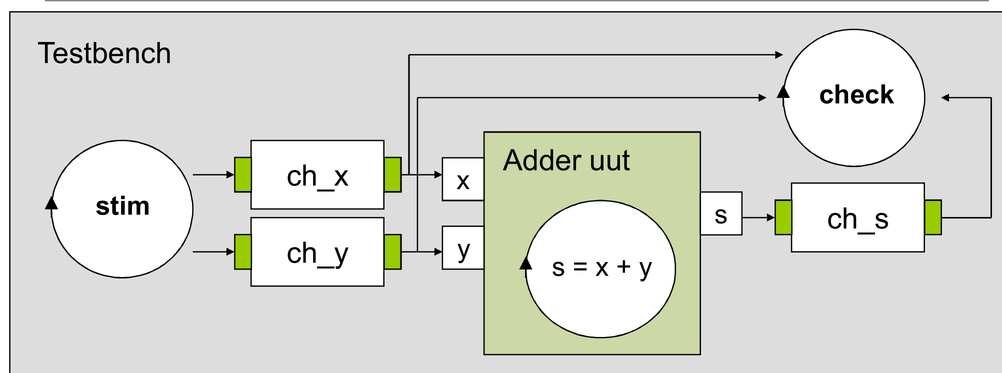
### SC_THREAD

- Started only once, at beginning of simulation
- May (and must) call **`wait()`**
- Often contains infinite loop
- May (and must) block – gives other processes chance to execute
- May use both non-blocking and blocking communications

*Source: M. Radetzki*



# SystemC Hierarchy (SC_MODULE)




```
SC_MODULE(Testbench)
{
  sc_signal<int> ch_x, ch_y, ch_s;   // channels & variables
  Adder uut;                         // submodule instance
  void stim();                       // stimuli process
  void check();                      // checking process
  ...
```

*Source: M. Radetzki*

## SC_MODULE Structure

```
class Testbench: public sc_module
{                                              // top level; no ports
  sc_signal<int> ch_x, ch_y, ch_s;             // channels
  Adder uut;                                   // Adder instance
  void stim();                                 // stimuli process
  void check();                                // checking process

  SC_HAS_PROCESS(Testbench);                   // needed if SC_CTOR not used
  Testbench(sc_module_name nm)                 // custom constructor
   : sc_module(nm), uut("uut"), ch_x("ch_x")   // initializer list
    SC_THREAD(stim);             // without sensitivity
    SC_METHOD(check);
    sensitive << ch_s;           // sensitivity for check
                                 // structural connectivity
    uut.x(ch_x);                 // port x of uut bound to ch_x
    uut.y(ch_y);                 // port y of uut bound to ch_y
    uut.s(ch_s);                 // port s of uut bound to ch_s
  }
};
```

Source: M. Radetzki



## Implementation of Module Processes

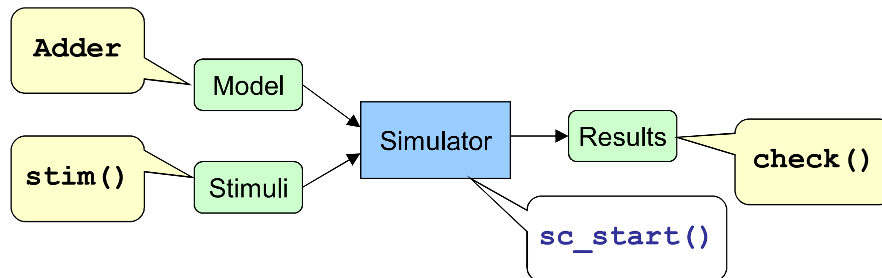
```
// file testbench.cpp
#include "testbench.h"

void Testbench::stim()  // SC_THREAD
{
  ch_x = 3; ch_y = 4;   // first stimulus
  wait(10, SC_NS);      // wait for 10 ns
  ch_x = 7; ch_y = 0;   // second stimulus
  wait(10, SC_NS);      // wait (no sensitivity!)
  ...                   // further stimuli
}

void Testbench::check() // SC_METHOD
{
  cout << ch_x << ch_y << ch_s << endl;   // debug output
  if( ch_s != ch_x+ch_y ) sc_stop();      // stop simulation
  else cout << "-> OK" << endl;
}
```

Source: M. Radetzki

## Invoking the Simulation from `sc_main`




```
// file main.cpp
int sc_main(int argc, char *argv[])  // cf. C++ main()
{
  Testbench tb("tb");         // Elaboration (constructors)
  sc_start();
  cout << "Simulation finished" << endl;
}
```

- Debugging: C++ debuggers (e.g. gdb/ddd)

*Source: M. Radetzki*



## Waveform Tracing

```
// file main.cpp
#include "testbench.h"
int sc_main(int argc, char *argv[])  // cf. C++ main()
{
  Testbench tb("tb");
  sc_trace_file *handle;  // file handle declaration
  handle = sc_create_vcd_trace_file("waveforms");
  sc_trace(handle, tb.ch_x, "ch_x");
  sc_trace(handle, tb.ch_y, "ch_y");
  sc_trace(handle, tb.ch_s,    "s");
  sc_start();
  cout << "simulation finished" << endl;
  sc_close_vcd_trace_file(handle);
}
```

tracing of channels ch_x, ch_y, and ch_s within instance tb

Label for waveform viewer

- VCD files, use any waveform viewer, e.g. gtkwave

*Source: M. Radetzki*

## Lecture 3: Outline

- ✓ System design languages
  - ✓ Goals, requirements
  - ✓ Separation of computation & communication
- The SystemC language
  - ✓ Basic SystemC model structure
  - ✓ Core language syntax
  - Data types
  - Events and simulation semantics
  - Channel library
  - Example



## Bit Data Types

| Type | bool (C++ type) | sc_logic |
|---|---|---|
| Values | false (0),<br>true (1) | '0', '1',<br>'X' (unknown),<br>'Z' (high-impedance) |
| Logic Operations | &&, \|\|, !, etc.<br>(see C++) | &, \|, ^, ~ |
| Assignment | = etc. (C++) | =, &=, \|=, ^= |
| Comparison | ==, != | ==, != |

```
Example: sc_logic a, b;
         a = '0';
         b = 'Z';
         c = a | b;   // result?
```

*Source: M. Radetzki*

## Vector Data Types

<table>
<tr><td>Type</td><td>sc_bv<N><br>vector of N bool</td><td>sc_lv<N><br>vector of N<br>sc_logic</td></tr>
<tr><td>Values</td><td>e.g. “01001100“</td><td>e.g. “01XZ0011“</td></tr>
<tr><td>Logic Operations</td><td colspan="2">~, &, |, ^, >>, <<</td></tr>
<tr><td>Assignment</td><td colspan="2">=, &=, |=, ^=</td></tr>
<tr><td>Comparison</td><td colspan="2">==, !=</td></tr>
<tr><td>Selection</td><td colspan="2">[int], range(int, int), (int, int)</td></tr>
<tr><td>Concatenation</td><td colspan="2">concat(vec), (vec, vec)</td></tr>
<tr><td>Arithmetic</td><td colspan="2">none</td></tr>
</table>

*Source: M. Radetzki*



## Arithmetic Data Types

<table>
<tr><td>Type</td><td>sc_int<N><br>vector of sign and<br>N-1 ≤ 63 binary<br>digits</td><td>sc_uint<N><br>vector of N ≤ 64<br>binary digits</td></tr>
<tr><td>Values</td><td>signed 2‘s compl.</td><td>unsigned</td></tr>
<tr><td>Logic Operations</td><td colspan="2">same as logic data types</td></tr>
<tr><td>Arithmetic Ops</td><td colspan="2">+, -, *, /, %, ++, --</td></tr>
<tr><td>Assignment</td><td colspan="2">=, &=, |=, ^=, +=, -=, *=, /=, %=</td></tr>
<tr><td>Comparison</td><td colspan="2">==, !=, >, <, <=, >=</td></tr>
<tr><td>Selection, Concat</td><td colspan="2">same as logic data types</td></tr>
</table>

**Note: can mix sc_int, sc_uint with C++ integer data types**
e.g.: sc_int + int is possible

*Source: M. Radetzki*

## Arbitrary Precision Arithmetics

| Type | sc_**big**int<N> vector of sign and N-1 binary digits | sc_**big**uint<N> vector of N binary digits |
|---|---|---|
| Values | signed 2's compl. | unsigned |
| Operations | same as sc_int, sc_uint | |

- **Notes**
  - can mix sc_bigint, sc_biguint with sc_int, sc_uint and C++ integer data types
  - precision is 64 bit if no big data type is involved in an expression
  - precision is arbitrary if big data type involved

*Source: M. Radetzki*



## Fixed Point Data Types

| Type | sc_fixed<…> | sc_ufixed<…> |
|---|---|---|
| Values | signed 2's complement | unsigned |
| Parameters | wl : total number of bits (word length)<br>iwl : bits before **.** (integer word length)<br>q_mode: quantization mode<br>o_mode: overflow mode, e.g. saturated<br>n_bits: (related to saturation) | |
| Arithmetic Ops | +, -, *, /, <<, >>, ++, -- | |
| Assignment | =, +=, -=, *=, /= | |
| Comparison | ==, !=, >, <, <=, >= | |

## Literals

- **Values of SystemC vector types can be written as:**

```
sc_lv<8> byte;
```

  - bitstrings

```
byte = "10101010";
byte = "1010XXXX";
```

  - binary string

```
byte = "0b10101010";
```

  - decimal string

```
byte = "0d170";
```

  - hex string

```
byte = "0xAA";  // what if "0X11"?
```

  - C++ number

```
byte = 170;
byte = 0xAA;
```

Source: M. Radetzki



## Events

- **Declaration:** `sc_event <name>;`
- **Immediate triggering:** `<name>.notify();`
- **Waiting for occurrence:** `wait( <name> );`

```
int x;
int y;
sc_event new_stimulus;

void Testbench::stim()
{
  x = 3; y = 4;
  new_stimulus.notify();
  x = 7; y = 0;
  new_stimulus.notify();
  // stimulus 7, 0 again
  new_stimulus.notify();
  ...
}
```

```
void Testbench::check()
{
  for(;;)
  {
    wait(new_stimulus);
    if( s == x+y )
      cout << "OK" << ...;
    else
      cout << "ERROR" << ...;
  }
}
```

Source: M. Radetzki

# Time

- `sc_time` data type
- Time units:
  - SC_FS femtosecond $10^{-15}$s
  - SC_PS picosecond $10^{-12}$s
  - SC_NS nanosecond $10^{-9}$s
  - SC_US microsecond $10^{-6}$s
  - SC_MS millisecond $10^{-3}$s
  - SC_SEC second $10^{0}$s
- Time object: `sc_time <name>(<magnitude>, <unit>);`
- e.g.: `sc_time delay(10, SC_NS);`
- usage, e.g.: `wait(delay);`
- alternative: `wait(10, SC_NS);`

Source: M. Radetzki



# Waiting on Events

```
sc_event a, b, c;
sc_time t(...);

wait();

wait(a);

wait(a & b & c);
wait(a | b | c);

wait(t);
wait(t, a & b);
```

**Static sensitivity**

`sensitive << ...`

(→ `wait();`)

**Dynamic sensitivity**

... on a single event (→ `wait(a);`)

... on a combination of events

- all events have happened (→ `wait(a & b & c);`)
- at least one event has happened (→ `wait(a | b | c);`)

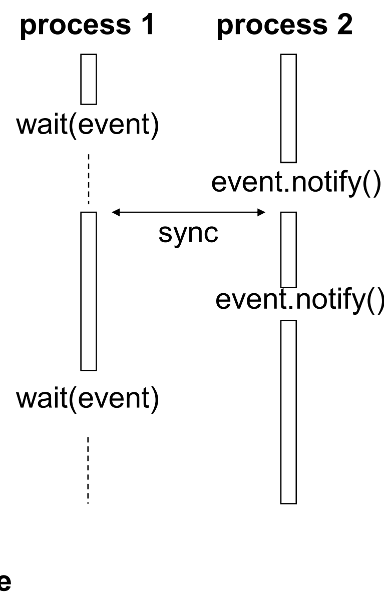
... for a time period (→ `wait(t);`)

... timeout (wait on event no longer than *t*) (→ `wait(t, a & b);`)

Source: M. Radetzki

## Event Semantics

- Process can wait for an event
- Event can be generated (triggered/notified) by another process
- Events are forgotten after being triggered
- Wait operation must have been invoked before a triggered event in order to "receive" that event

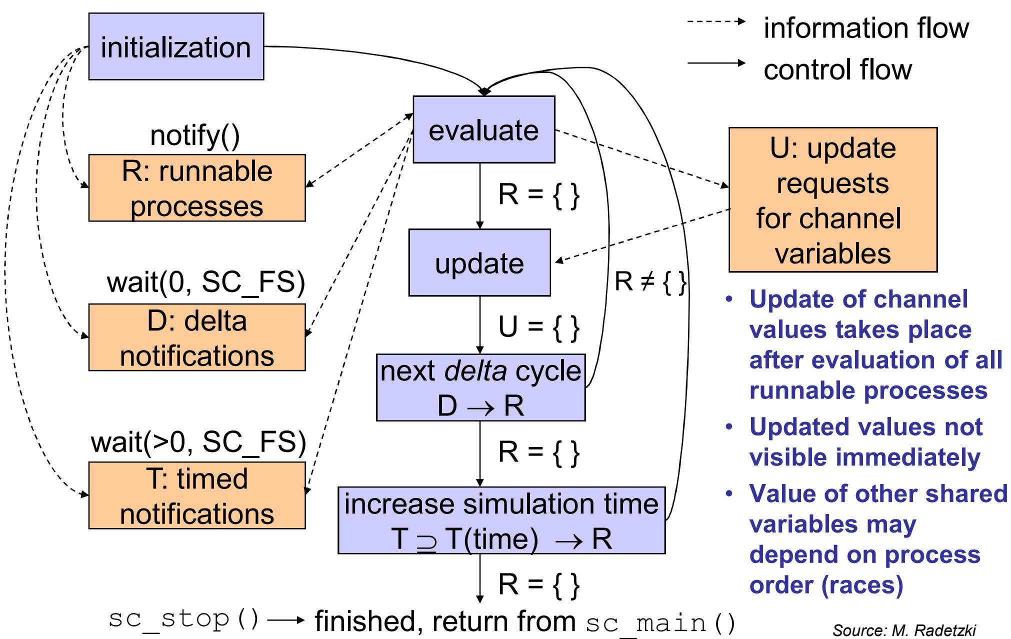



waiting active

Source: M. Radetzki



## Simulation Cycle




- Update of channel values takes place after evaluation of all runnable processes
- Updated values not visible immediately
- Value of other shared variables may depend on process order (races)

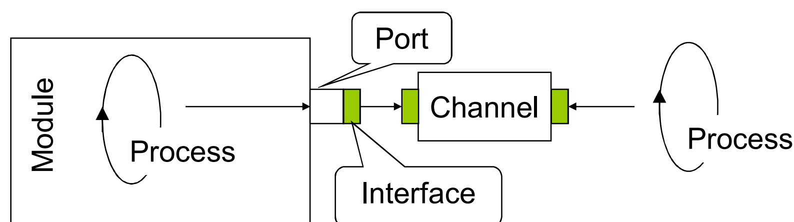
Source: M. Radetzki

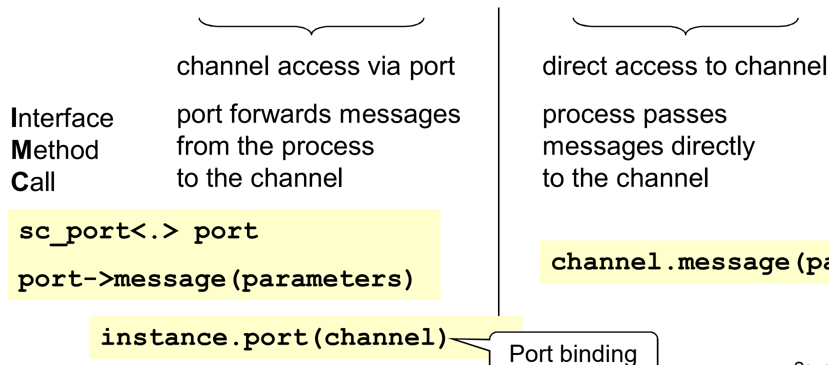
# Lecture 3: Outline

- ✓ System design languages
  - ✓ Goals, requirements
  - ✓ Separation of computation & communication
- The SystemC language
  - ✓ Basic SystemC model structure
  - ✓ Core language syntax and semantics
  - ✓ Data types
  - ✓ Events and simulation semantics
  - Channel library
  - Example



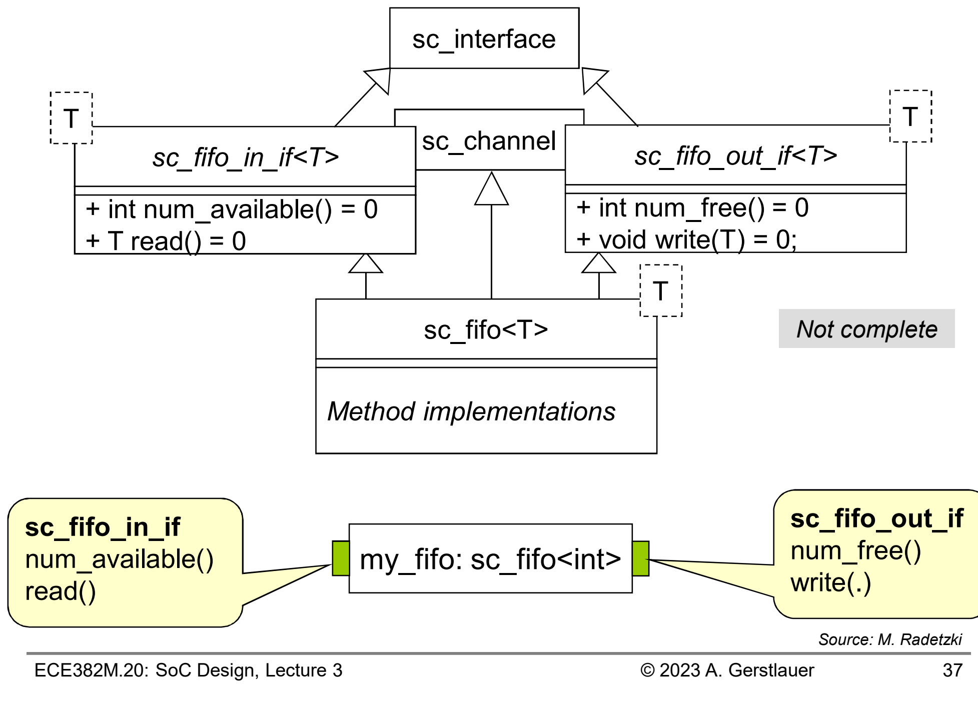
# SystemC Channels



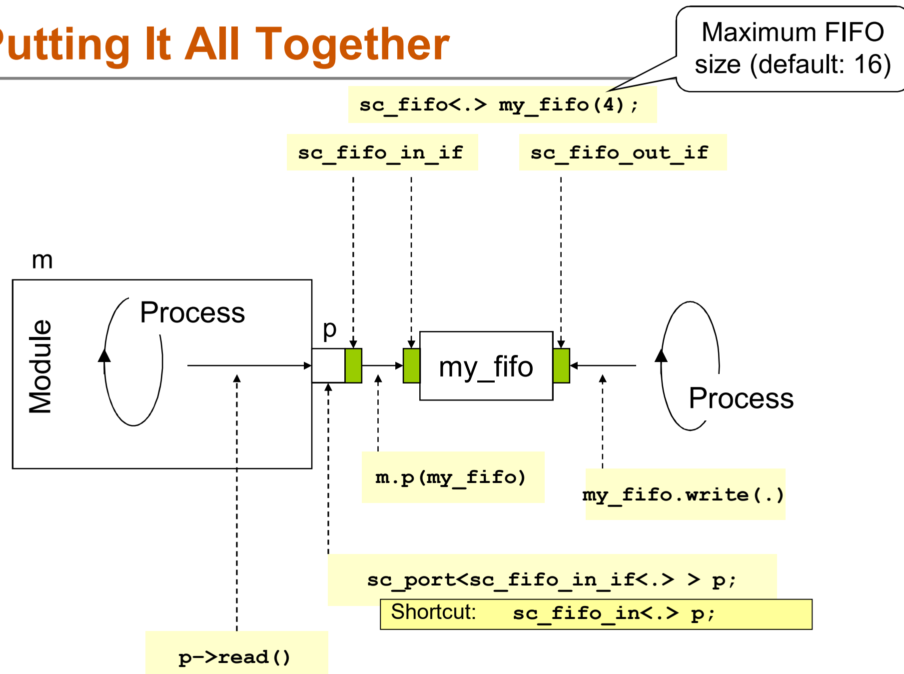

Source: M. Radetzki

## Channels & Interfaces (sc_fifo)

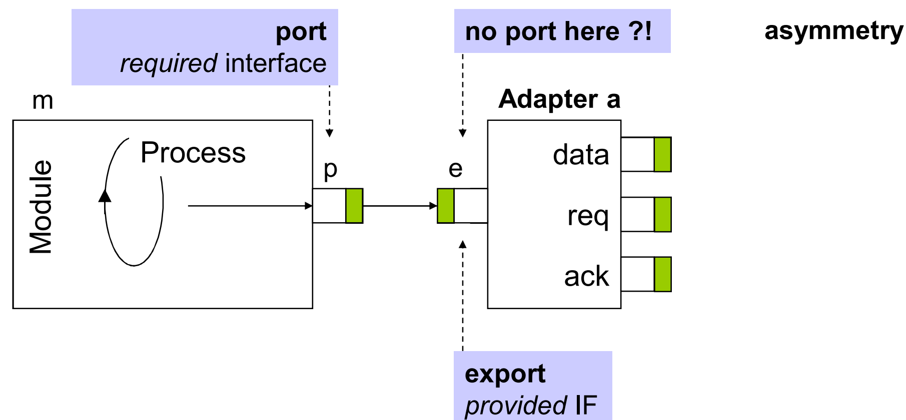





## Putting It All Together



## Exports




connection / binding:
`m.p( a.e )`

alternative syntax:
`m.p.bind(a.e)`

`sc_export<sc_fifo_in_if<T> > e;`

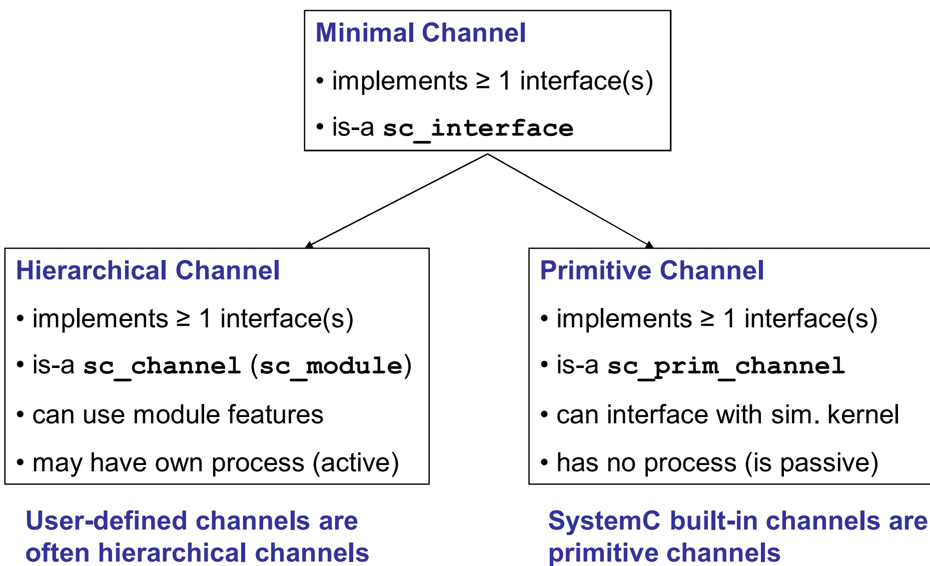
In constructor:
`e(*this);` or `e.bind(*this);`

- **Alternative to sc_export is to inherit from interface (half channel)**
- **sc_modules other than adapters may have exports**

*Source: M. Radetzki*



## SystemC Channels




**User-defined channels are often hierarchical channels**

**SystemC built-in channels are primitive channels**

*Source: M. Radetzki*

## SystemC Built-In Channels

| Channel | Matching Ports (Shortcuts) | Events |
|---|---|---|
| sc_signal<T> | sc_in<T><br>sc_out<T><br>sc_inout<T> | value changed |
| sc_buffer<T> | sc_in<T><br>sc_out<T><br>sc_inout<T> | value written (also if same as previous value) |
| sc_fifo<T> | sc_fifo_in<T><br>sc_fifo_out<T> | fifo contents changed |
| sc_semaphore | -- | -- |
| sc_mutex | -- | -- |
| sc_clock | sc_in<bool> | value changed |



## Custom FIFO Example: Channel

```
class write_if :
        virtual public sc_interface
{
   public:
     virtual void write(char) = 0;
     virtual void reset() = 0;
};

class read_if :
         virtual public sc_interface
{
   public:
     virtual void read(char &) = 0;
     virtual int num_available() = 0;
};
```

```
SC_MODULE(fifo),
            public write_if, public read_if
{
   public:
     SC_CTOR(fifo):
                       num_elements(0), first(0) {}

     void write(char c) {
       if (num_elements == max)
         wait(read_event);

       data[(first + num_elements++) % max] = c;
       write_event.notify();
     }

     void read(char &c){
       if (num_elements == 0)
         wait(write_event);

       c = data[first]; --num_elements;
       first = (first + 1) % max;
       read_event.notify();
     }

     void reset() { num_elements = first = 0; }
     int num_available() { return num_elements; }

   private:
     enum e { max = 10 };
     char data[max];
     int num_elements, first;
     sc_event write_event, read_event;
};
```

*Source: S. Swan, Cadence Design Systems*

## Custom FIFO Example: Modules

```
SC_MODULE(producer)
{
  public:
    sc_port<write_if> out;

    SC_CTOR(producer)
    {
      SC_THREAD(main);
    }

    void main()
    {
      char c;

      while (true) {
        …
        out->write(c);
        if (…) out->reset();
      }
    }
};
```

```
SC_MODULE(consumer)
{
  public:
    sc_port<read_if> in;

    SC_CTOR(consumer)
    {
      SC_THREAD(main);
    }

    void main()
    {
      char c;

      while (true) {
        in->read(c);
        if (in->num_available())
          …
      }
    }
};
```

*Source: S. Swan, Cadence Design Systems*



## Custom FIFO Example: Main

```
SC_MODULE(top),
{
  public:
    fifo *fifo_inst;
    producer *prod_inst;
    consumer *cons_inst;

    SC_CTOR(top)
    {
      fifo_inst = new fifo("Fifo1");

      prod_inst = new producer("Producer1");
      prod_inst->out(*fifo_inst);

      cons_inst = new consumer("Consumer1");
      cons_inst->in(*fifo_inst);
    }
};

int sc_main (int argc , char *argv[])
{
   top top1("Top1");
   sc_start();
   return 0;
}
```

*Source: S. Swan, Cadence Design Systems*

## Lecture 3: Summary

- **SystemC language**
  - C++ class library
    - Don’t invent a new language, leverage existing tools
  - De-facto industry-standard
    - SoC architecture & virtual platform modeling
- **SystemC class library**
  - Discrete event based simulation kernel
    - Concurrency, events and logical time
  - Core library
    - Modules and channels
    - Threads, processes
    - Data types
  - Standard channel library