SystemC Compilation, Execution and Debugging

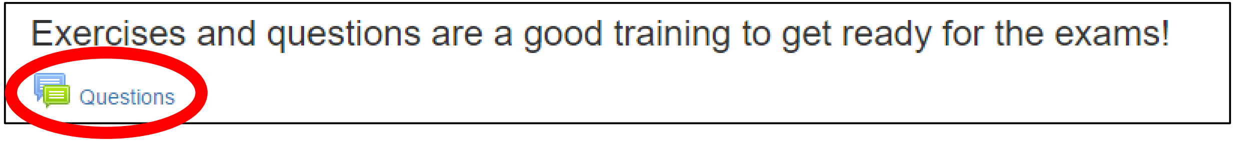
Stefano Spellini

Course Assignment

- Produce a **report** for **every** laboratory class
  - Containing a **discussion** of the solutions of the proposed exercises
  - If requested, provide also the **code**
  - **Each lesson's report will be composed in a single course report, to be delivered on or before the end of the academic year**
    - **No intermediate deadlines, NO PARTIAL REPORTS WILL BE ACCEPTED**
  - **Max 3 points**
    - Only **within the course (February 2021)**: 1 bonus point!
  - **Remember: the assignment is valid only until the next edition of the course**
    - **Last chance in September 2021**
    - After September 2021:
      - assignment following new course directives
      - With a new lecturer (maybe new rules)

Support

- On the new e-learning portal:
  - Support material
  - Forum for Q&A



Exercises and questions are a good training to get ready for the exams!

Questions

- Dott. Spellini’s office hour:
  - Ca’ Vignal 2, Floor 1, Room 71
  - **Always “on demand”** after arranging a meeting **via email!**
    - **2 working days in advance:**
      - **Ask Thursday for Monday, Ask Monday for Wednesday**
      - **UNIVR emails only, addressed to:**

`stefano.spellini@univr.it`

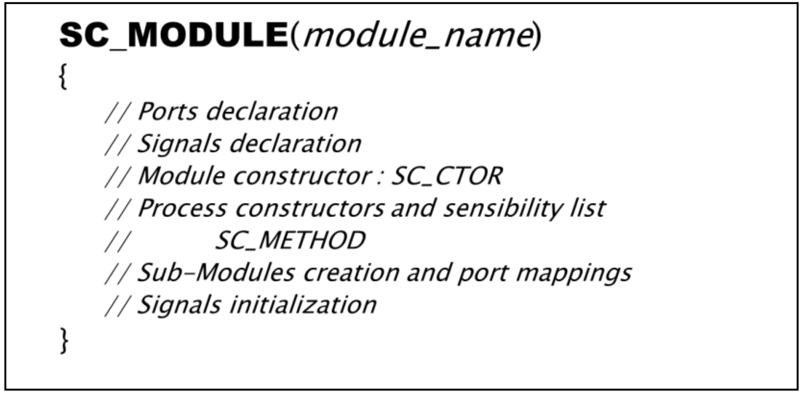
SystemC highlights

- Supports hardware and software *co-design*
- Even Better: supports Electronic Level Design!
- Developing an *executable specification* avoids inconsistencies and errors
  - Avoids wrong interpretation of the specification
  - Rapid prototyping
    - For verification, testing, performances evaluation, etc…
- SystemC has a rich set of *data types* to model the system
- It allows *multiple abstraction levels*
  - From high level design (e.g., TLM) down to cycle-accurate RTL level
- It is now capable of representing **multi-domain systems**
- Practical note: it defines a set of Macros…
  - It's better using the expanded expressions
  - This lesson uses Macros, expanded expressions are in the Language Reference Manual (LRM)

SYSTEMC MODELING

Modules

- Modules
  - Basic building blocks to partition a design
  - Fundamentally, C++ classes
  - Enable the partitioning of complex systems into smaller components
  - Hide internal data representation, using interfaces
    - Contain ports, signals, local data, other modules, processes and constructors

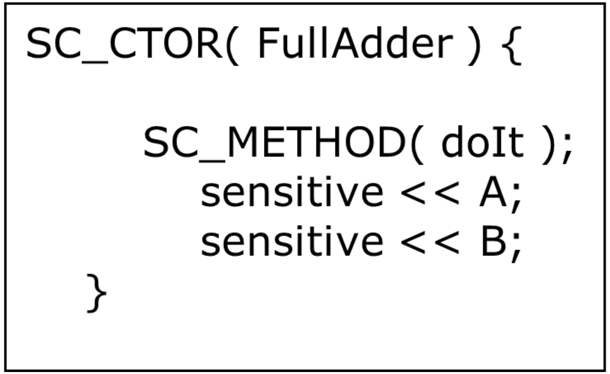


```
SC_MODULE(module_name)
{
     // Ports declaration
     // Signals declaration
     // Module constructor : SC_CTOR
     // Process constructors and sensibility list
     //          SC_METHOD
     // Sub-Modules creation and port mappings
     // Signals initialization
}
```

Module constructor

- Module constructor
  - Initializes and declares all processes contained in the module and the rules to activate them
  - Initializes variables to default or user defined values

Example: Full Adder constructor



```
SC_CTOR( FullAdder ) {

        SC_METHOD( doIt );
            sensitive << A;
            sensitive << B;
   }
```

Hierarchy: sub-modules instantiation

- Sub-modules instantiation:
  - Instantiate module

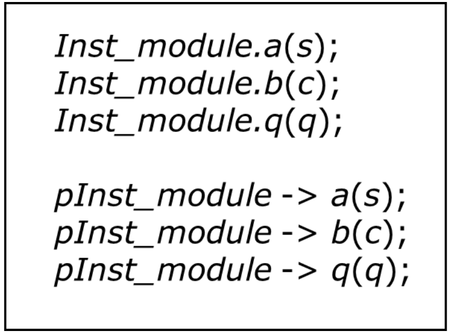
    Module_type Inst_module (“label”);
  - Instantiate module as a pointer

    Module_type *pInst_module;

    // Instantiate at the module constructor SC_CTOR

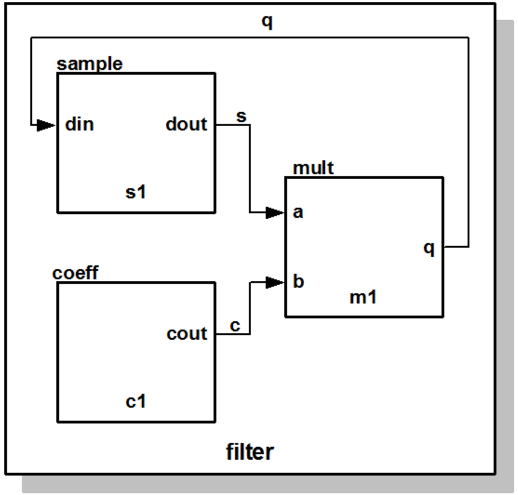
    pInst_module = new module_type (“label”);

- Connection of sub-modules
  - Named connection



```
Inst_module.a(s);
Inst_module.b(c);
Inst_module.q(q);

pInst_module -> a(s);
pInst_module -> b(c);
pInst_module -> q(q);
```

Hierarchy Example



```
SC_MODULE(filter) {
        // Sub-modules : "components"
        sample *s1;
        coeff *c1;
        mult *m1;

        sc_signal<sc_uint 32> > q, s, c; // Signals

        // Constructor : "architecture"
        SC_CTOR(filter) {

                // Sub-modules instantiation and
            mapping
                s1 = new sample ("s1");
                s1->din(q); // named mapping
                s1->dout(s);

                c1 = new coeff("c1");
                c1->out(c); // named mapping

                m1 = new mult ("m1");
                (*m1)(s, c, q); // Positional mapping

          }
}
```

Internal Data Storage

- Local variables
  - Can not be used to connect ports
- Allowed data types
  - C++ types
  - SystemC types
  - User defined types

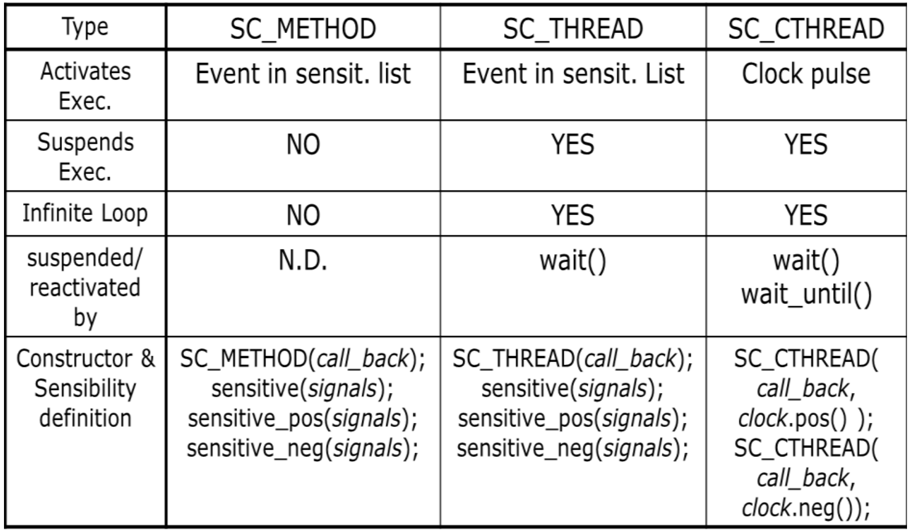
```
SC_MODULE( Mux21 ) {

    sc_in< sc_uint<8> >   in1;
    sc_in< sc_uint<8> >   in2;
    sc_in< bool >       selection;
    sc_out< sc_uint<8> >  out;

    void doIt( void );

    SC_CTOR( Mux21 ) {

        SC_METHOD( doIt );
            sensitive << selection;
            sensitive << in1;
            sensitive << in2;

    }

};
```

Processes

- Processes are functions that are identified into the SystemC kernel
  - Processes are very similar to C++ functions or methods
  - Processes implement the functionality of modules
  - They are called if one signal of the sensitivity list changes its value

- Processes can be Methods, Threads and CThreads

- Methods
  - When activated, executes and returns
    - SC_METHOD(process_name)
- Threads
  - Can be suspended and reactivated
    - wait() -> suspends
    - one sensitivity list event -> activates
    - SC_THREAD(process_name)
- CThreads
  - Are activated in the clock pulse
    - SC_CTHREAD(process_name, clock value);
  - Deprecated



Type	SC_METHOD	SC_THREAD	SC_CTHREAD
Activates Exec.	Event in sensit. list	Event in sensit. List	Clock pulse
Suspends Exec.	NO	YES	YES
Infinite Loop	NO	YES	YES

suspended/ reactivated by	N.D.	wait()	wait()
			wait_until()

Constructor & Sensibility definition	SC_METHOD(*call_back*);
	sensitive(*signals*);
	sensitive_pos(*signals*);
	sensitive_neg(*signals*);
		SC_THREAD(*call_back*);
		sensitive(*signals*);
		sensitive_pos(*signals*);
		sensitive_neg(*signals*);
			SC_CTHREAD(*call_back*, *clock*.pos());
			SC_CTHREAD(*call_back*, *clock*.neg());

Process example

Into the .H file

```
void doIt( void );

    SC_CTOR( Mux21 ) {

        SC_METHOD( doIt );
            sensitive << selection;
            sensitive << in1;
            sensitive << in2;

    }
```

Into the .CPP file

```
void Mux21::doIt( void ) {

    sc_uint<8> out_tmp;

    if( selection.read() ) {
        out_tmp = in2.read();
    } else {
        out_tmp = in1.read();
    }

    out.write( out_tmp );
}
```

Ports and Signals

- Ports of a module are the external interfaces that pass information to and from a module
  - In SystemC one port can be IN, OUT or INOUT
  - Implementing the read( ) and write( ) methods
- Signals are used to connect module ports allowing modules to communicate
- Types of ports and signals:
  - All natives C/C++ types
  - All SystemC types
  - User defined types
- How to declare
  - Input: **sc_in<port_type>**
  - Output: **sc_out<port_type>**
  - Bi-Directional : **sc_inout<port_type>**

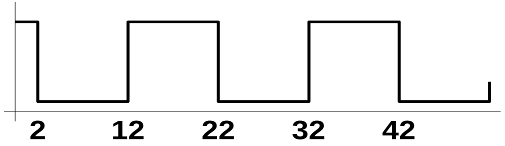
Clock

- Special object

  sc_clock clock_name("clock_label", period, duty_ratio, offset, initial_value);

- Clock connection
  - **f1.clk(clk_signal);**

```
sc_clock clock1 ("clock1", 20, 0.5, 2, true);
```



- Better avoiding using the sc_clock object!
  - It is **preferable** to define a signal implementing the clock
  - Many alternatives for the data-type
    - **Bool**
    - **sc_logic**
    - **sc_bit** (inefficient!)
  - How to update the signal?
    - Temporized thread with no sensitivity list!

Data types

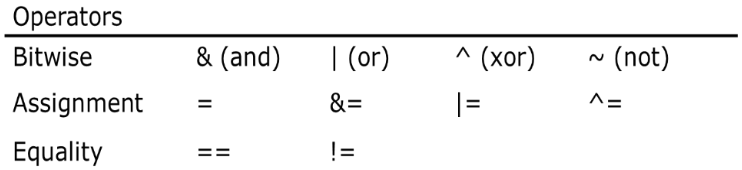
- SystemC supports:
  - C/C++ native types
  - SystemC types
- SystemC types
  - Types for systems modelling
  - 2 values (‘0’,’1’)
  - 4 values (‘0’,’1’,’Z’,’X’)
  - Arbitrary size integer (Signed/Unsigned)
  - Fixed point types
- Domain-specific data-types
  - Data-types for analog models
    - Electrical signals
    - Linear Signal Flow signals
  - Data-types for ESL descriptions
    - Socket
    - Payloads

Type	Description
sc_logic	Simple bit with 4 values(0/1/X/Z)
sc_int	Signed Integer from 1-64 bits
sc_uint	Unsigned Integer from 1-64 bits
sc_bigint	Arbitrary size signed integer
sc_biguint	Arbitrary size unsigned integer
sc_bv	Arbitrary size 2-values vector
sc_lv	Arbitrary size 4-values vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

SystemC data types: 2-value and multi-value logics

- **SC_BIT** type
  - Simple bit type
    - Declaration
      - **bool my_bit;**
    - Assignment similar to char
      - **my_bit = ‘1’;**

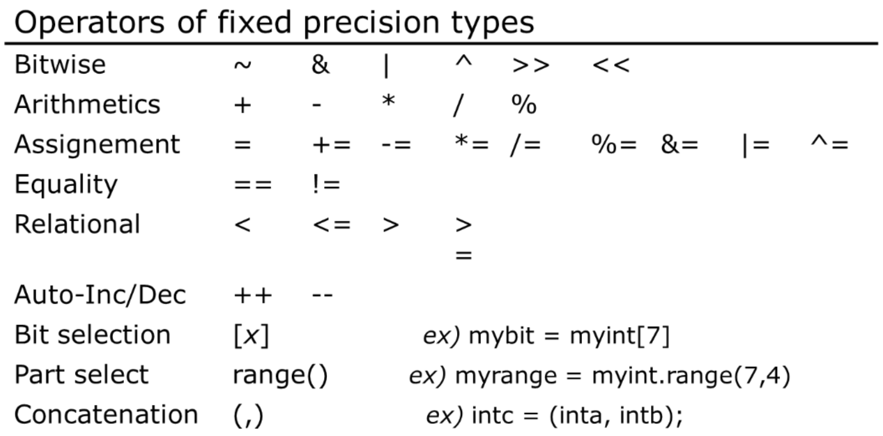
- **SC_LOGIC** type
  - More general than bool, 4 values
    - ‘0’ (false)
    - ‘1’ (true)
    - ‘X’ (undefined)
    - ‘Z’ (high-impedance)
  - Declaration
    - **sc_logic my_logic;**
  - Assignment like bool
    - **my_logic = ‘0’;**
    - **my_logic = ‘Z’;**
  - Operators like bool
  - Higher simulation times than bool



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

SystemC data types: Integers

- Fixed precision integers
  - Used when arithmetic operations need fixed size arithmetic operands
  - INT can be converted in UINT and vice-versa
  - 1-64 bits integer
    - **sc_int<n>** -- signed integer with n-bits
    - **sc_uint<n>** -- unsigned integer with n-bits



Operators of fixed precision types

Bitwise	~	&	|	^	>>	<<			
Arithmetics	+	-	*	/	%				
Assignement	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto-Inc/Dec	++	--							
Bit selection	[x]			*ex)* mybit = myint[7]					
Part select	range()			*ex)* myrange = myint.range(7,4)					
Concatenation	(,)			*ex)* intc = (inta, intb);					

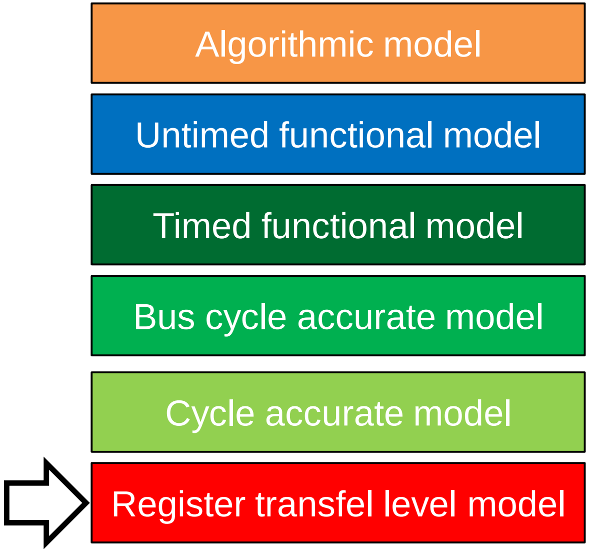
SystemC data types: Vectors

- Bit vector
  - **sc_bv<n>**
  - 2-value vector (0/1)
  - Not used in arithmetics operations
  - Faster simulation than sc_lv

- Logic Vector
  - **sc_lv<n>**
  - Vector to the sc_logic type
  - Assignment operator (“=“)
  - my_vector = “XZ01”
  - Conversion between vector and integer (int or uint)
  - Assignment between sc_bv and sc_lv

Register Transfer Level

- Describe the operation of a synchronous digital circuit
  - Flow of signals
    - Transfer of data between registers
  - Logical operations performed on signals
- Managed by a controller
- Synchronized by a clock



Algorithmic model

Untimed functional model

Timed functional model

Bus cycle accurate model

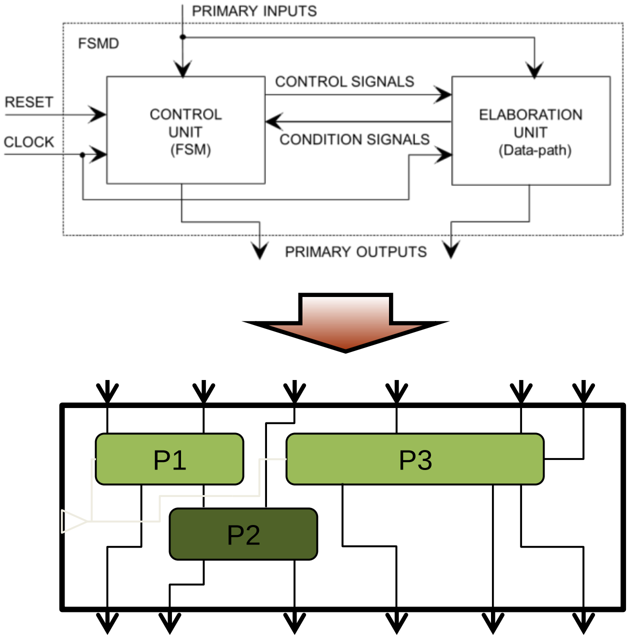
Cycle accurate model

Register transfel level model

Register Transfer Level (Cont.d)

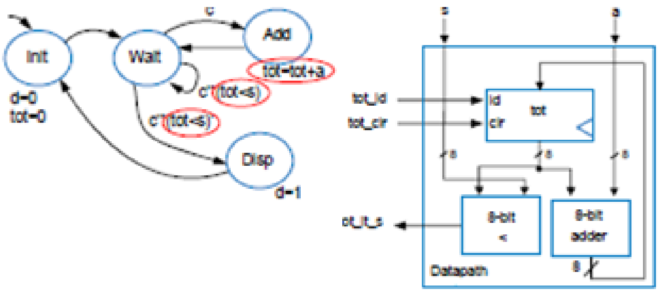
- Interconnection of FSM + Data-path
  - Control/condition signals are identified
  - FSM is represented by states and transitions
  - Data-path is represented by registers, multiplexers and operators

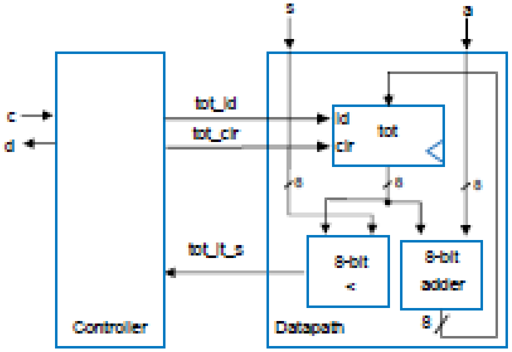
- Automatic translation into a set of registers and library components (RTL)

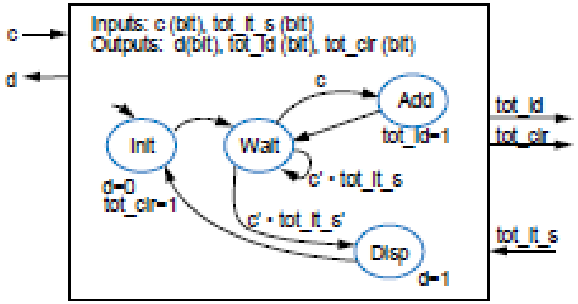


How to build you RTL code

1. Describe the behavior with a high-level FSM
2. Determine the datapath
   - What shall I do when the FSM is in a certain state?
3. Connections between datapath and FSM
4. Determine the controller FSM







SystemC representation of FSMDs

- One process for computing the next state and updating outputs
  - FSMD representation

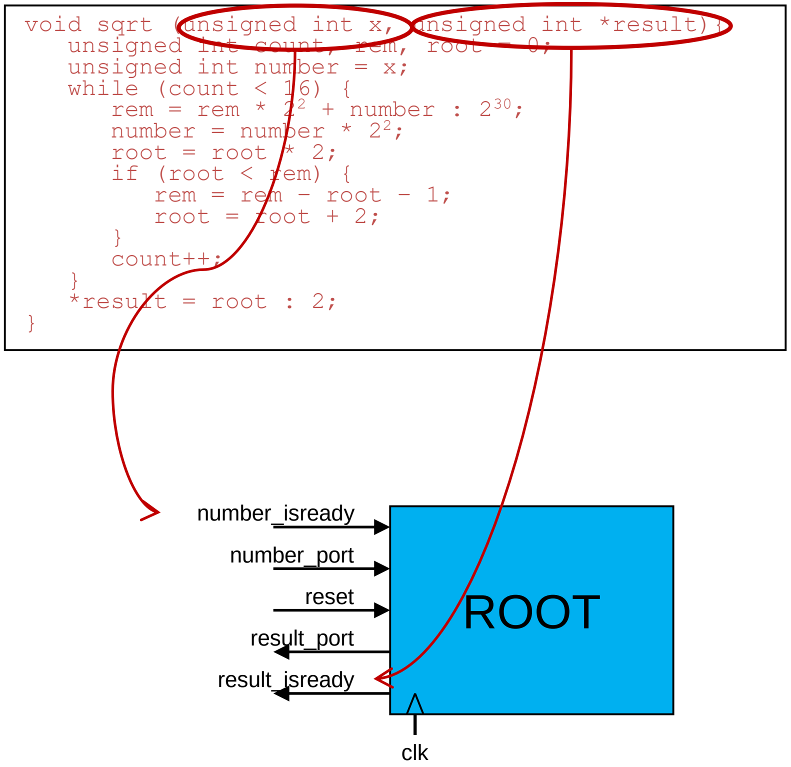
Merges controller and datapath in a FSMD representation

- Two processes
  - Compute the next state and outputs
  - Update the current state
- Three processes
  - Compute the next state
  - Compute the outputs
  - Update the current state

Divide controller and datapath

The *root* Module

- Interface definition
  - Allow interaction with
    the other modules
  - Input signals
    - number_is ready
    - number_port
    - result_isready
    - result_port
    - reset
    - clk

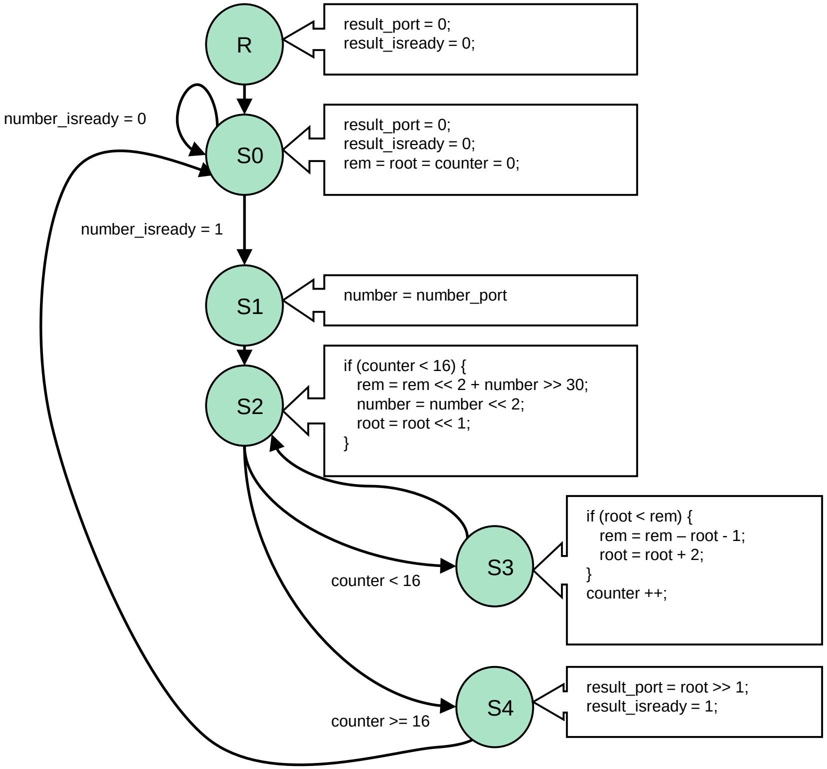


```
void sqrt (unsigned int x, unsigned int *result){
   unsigned int count, rem, root = 0;
   unsigned int number = x;
   while (count < 16) {
      rem = rem * 2^2 + number : 2^30;
      number = number * 2^2;
      root = root * 2;
      if (root < rem) {
         rem = rem - root - 1;
         root = root + 2;
      }
      count++;
   }
   *result = root : 2;
}
```

The *root* Module (Cont.d)

- Implementation based on a 6 states EFSM:
  - R: reset state
  - S0: initialization
  - S1: input reading
  - S2 – S3: computation
  - S4: Output writing

- Implementated with two processes
  - Internal variables and next state
  - Current state and outputs



```
result_port = 0;
result_isready = 0;
```

```
result_port = 0;
result_isready = 0;
rem = root = counter = 0;
```

```
number = number_port
```

```
if (counter < 16) {
  rem = rem << 2 + number >> 30;
  number = number << 2;
  root = root << 1;
}
```

```
if (root < rem) {
  rem = rem – root - 1;
  root = root + 2;
}
counter ++;
```

```
result_port = root >> 1;
result_isready = 1;
```

SystemC Simulation

SystemC Scheduler

- C++ is not suited for HW modeling
  - Does not express concurrent activities intrinsic of HW
  - Delays, time and clock are not provided
  - Data types are too far from HW representation
  - HW communication mechanisms (e.g., signals or ports) are different from SW communication mechanisms

- SystemC extends C++ to overcome such limitations
  - Simulation kernel
    - Controls simulation
    - Advancing of time
    - Making processes runnable
    - Update ports and signals
    - Notify events
    - …

Event based simulation

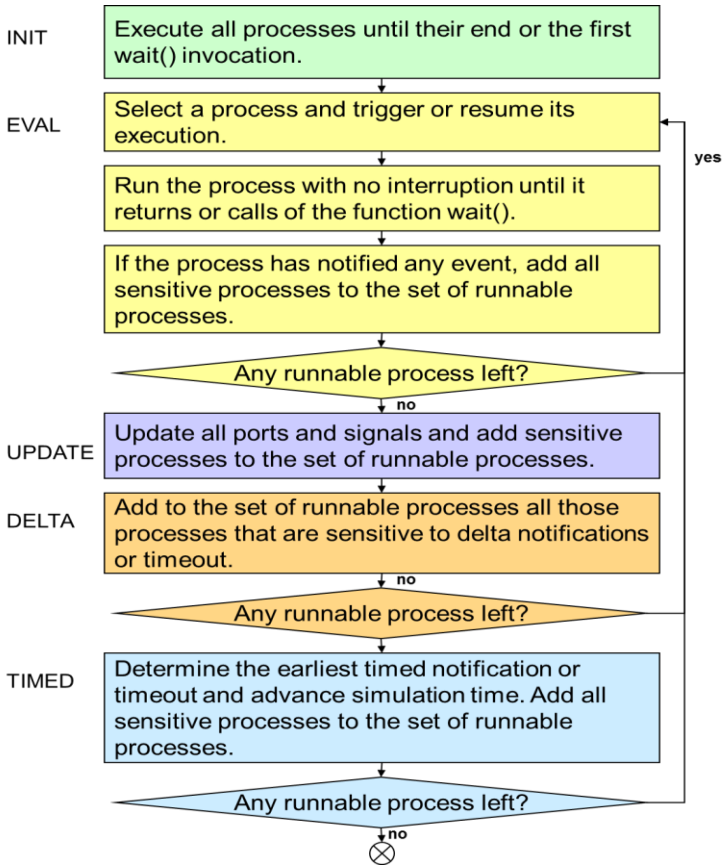
- SystemC simulation is event based
  - Events rule the evolution of the system
  - Processes are executed in response to the occurrence of events

- Scheduler
  - Uses events to prepare the *ready queue*
    - Processes that can be run at any time
  - Chooses the process that must run first
  - Whenever a process creates any events, sensitive processes are added to the ready queue
    - Executed at a later time

Event based simulation (Cont.d)

- Event types:
  - **Sc_event**
    - Wake up all processes sensitive to that event
  - Port/signal update
    - Awakens processes sensitive to that port or signal
  - Delta notification and timed notification
    - **Delta**: processes are made runnable in the subsequent evaluation phase
    - **Timed**: Processes are activated after the specified amount of simulation time
  - **Clock**
    - Makes runnable clocked processes
    - Allows to model the passing of the time
  - **Note: variables do not generate any event!**

SystemC scheduler



Execute all processes until their end or the first wait() invocation.

Select a process and trigger or resume its execution.

Run the process with no interruption until it returns or calls of the function wait().

If the process has notified any event, add all sensitive processes to the set of runnable processes.

Any runnable process left?

Update all ports and signals and add sensitive processes to the set of runnable processes.

Add to the set of runnable processes all those processes that are sensitive to delta notifications or timeout.

Any runnable process left?

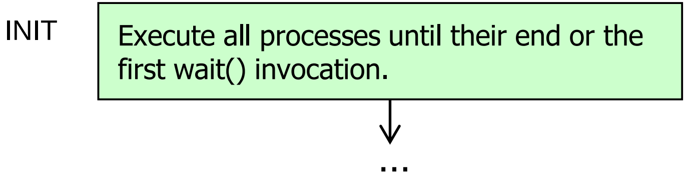
Determine the earliest timed notification or timeout and advance simulation time. Add all sensitive processes to the set of runnable processes.

Any runnable process left?

SystemC scheduler (Cont.d)

- Elaboration phase
  - Build internal data structures
  - Set up module hierarchy
  - Port binding

- Initialization phase
  - Executes all processes (SC_METHODs or SC_THREADs) until their end or their first wait() statement
  - Builds the first runnable queue

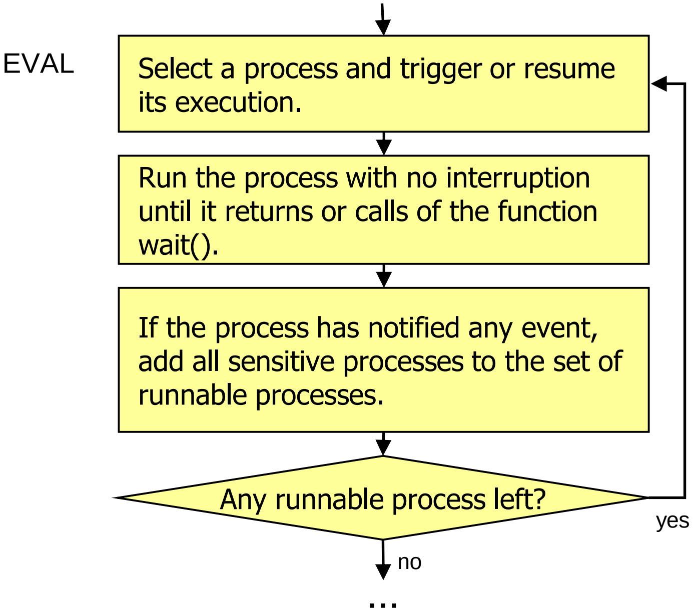

INIT

Execute all processes until their end or the first wait() invocation.

…

SystemC scheduler (Cont.d)

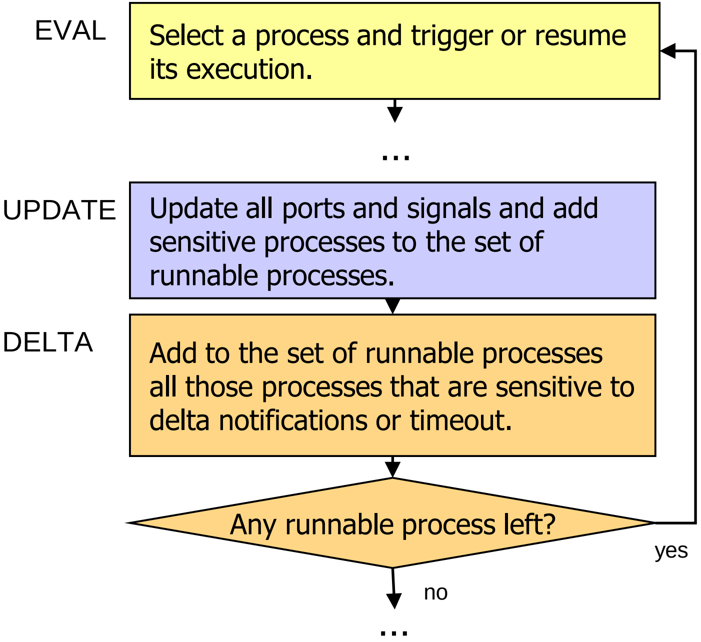
- Evaluation phase
  - Execution of all processes in the runnable set
  - Co-operative multitasking
    - Execute only one process at one time until process yelds control of the CPU
    - Processes can not pre-empt or interrupt other processes



EVAL
Select a process and trigger or resume its execution.
Run the process with no interruption until it returns or calls of the function wait().
If the process has notified any event, add all sensitive processes to the set of runnable processes.
Any runnable process left?
yes
no
...

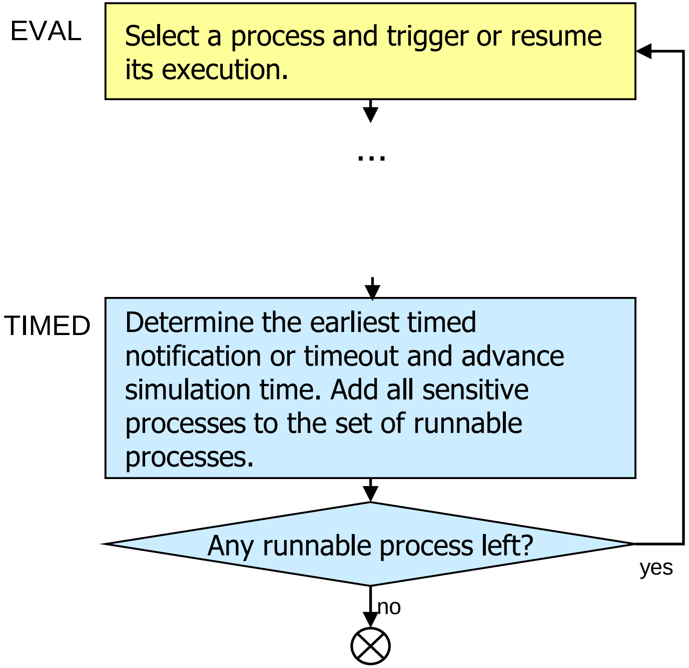
SystemC scheduler (Cont.d)

- Update phase
  - Update the value of ports and signals
  - All processes are added to the runnable queue
- Delta notification phase
  - If there is any delta notification or timeout, add processes to the runnable queue


EVAL
Select a process and trigger or resume its execution.
…
UPDATE
Update all ports and signals and add sensitive processes to the set of runnable processes.
DELTA
Add to the set of runnable processes all those processes that are sensitive to delta notifications or timeout.
Any runnable process left?
yes
no
…

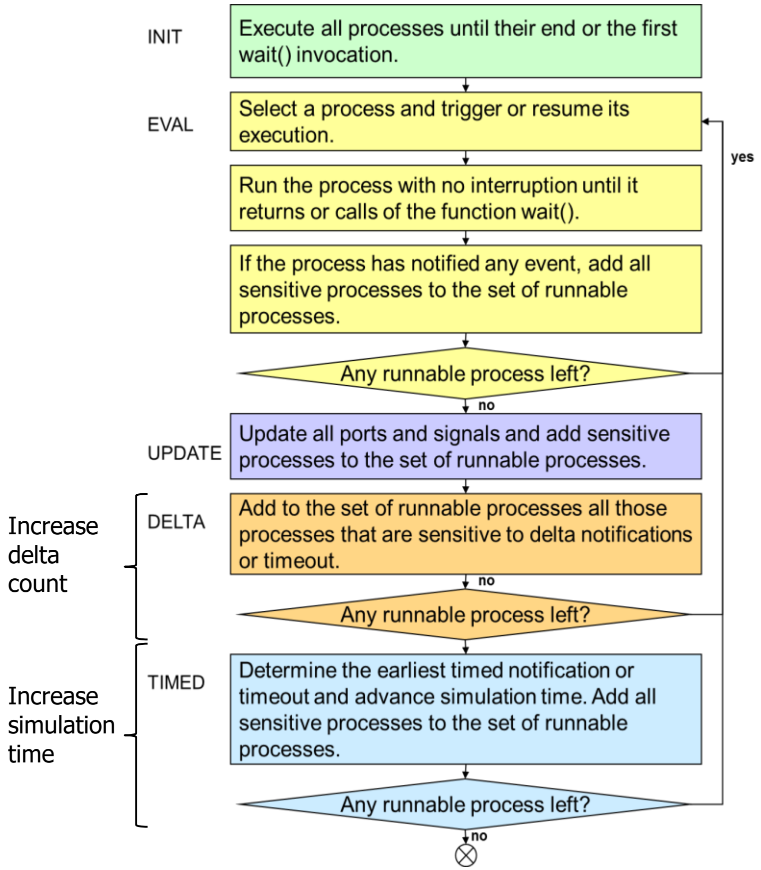
SystemC scheduler (Cont.d)

- Timed notification phase
  - Simulation time is advanced until the earliest of the next timed notification or timeout
    - Clock change
  - If there is no timed event, simulation is finished


EVAL
Select a process and trigger or resume its execution.
...
TIMED
Determine the earliest timed notification or timeout and advance simulation time. Add all sensitive processes to the set of runnable processes.
Any runnable process left?
yes
no

Time in SystemC

- **sc_time**
  - Used to represent simulation time and time intervals
    - Double + **sc_time_unit**
- **sc_time_stamp**
  - Return current simulation time
- **sc_delta_count**
  - Incremented once per delta cycle


INIT
Execute all processes until their end or the first wait() invocation.
EVAL
Select a process and trigger or resume its execution.
Run the process with no interruption until it returns or calls of the function wait().
If the process has notified any event, add all sensitive processes to the set of runnable processes.
Any runnable process left?
yes
no
UPDATE
Update all ports and signals and add sensitive processes to the set of runnable processes.
Increase delta count
DELTA
Add to the set of runnable processes all those processes that are sensitive to delta notifications or timeout.
no
Any runnable process left?
Increase simulation time
TIMED
Determine the earliest timed notification or timeout and advance simulation time. Add all sensitive processes to the set of runnable processes.
Any runnable process left?
no

Tracing signals

- Trace file
  - Record a timed-ordered sequence of value changes during simulation
  - How to:
    - Declare a file pointer variable
    - Open the VCD file for writing (**sc_create_vcd_trace_file**)
    - Add the signals you are interested in (**sc_trace** function)
    - Run your simulation (**main** function)
    - Close the file (**sc_close_vcd_trace_file**)

```
sc_trace_file *fp;
fp=sc_create_vcd_trace_file
     ("wave");
sc_trace(fp,clk,"clk");
sc_trace(fp,din,"din");
sc_trace(fp,dout,"dout");
sc_start(100, SC_NS);
sc_close_vcd_trace_file(fp);
```

Tracing signals (Cont.d)

- The VCD file is organized as follows:
  - Timescale

```
$timescale 1ps
```

  - List of traced signals
    - Each signal is assigned an identifier that substitutes the signal name all along the VCD file

```
$var wire    1  aac  NUMBER_READY       $end
```

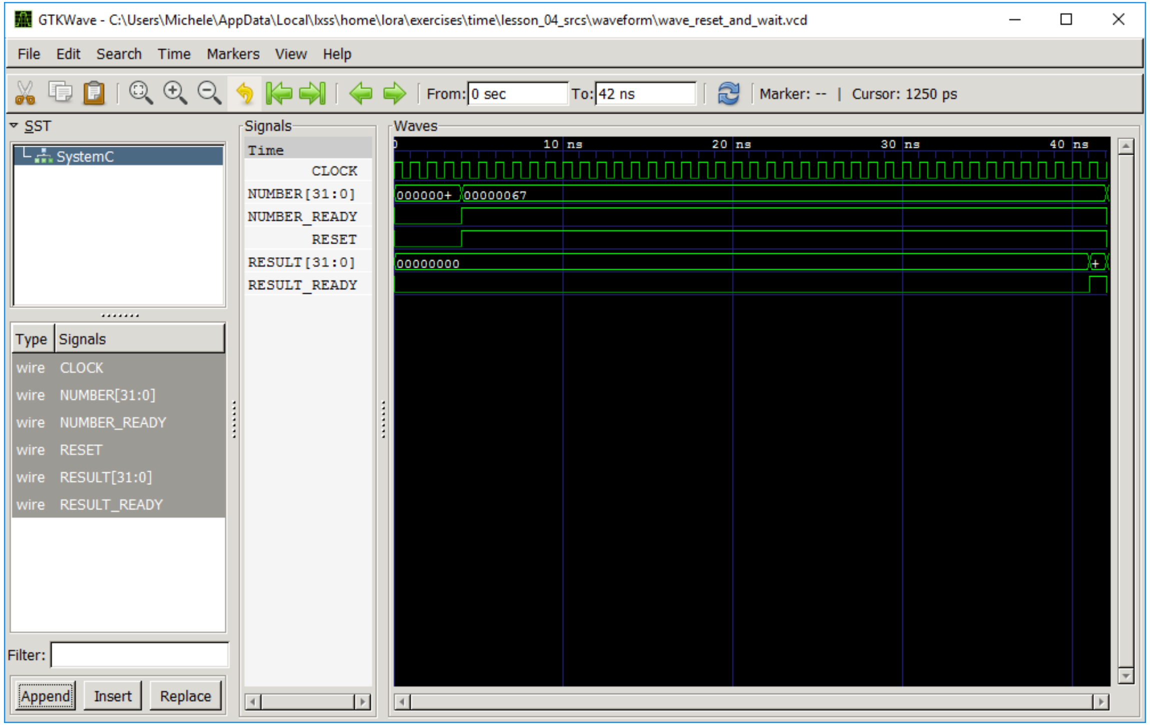
  - Whenever a signal is changed, the VCD file reports time and list of changed signals

```
#4000
1aac
```

- Trace files viewers
  - **GTKWave**
    - Download it on your machine: http://gtkwave.sourceforge.net/
    - Multiplatform: Mac OSX, Linux 32/64, Win32, source code…
    - Easy to use GUI
  - Modelsim (later in the course)
  - Alternative tools (both free and proprietary)
    - https://www.edaplayground.com/home **(Nice tool: take a look at it)**
    - Dinotrace
    - EZWave
    - Etc…

Tracing signals (Cont.d)

- GTKWave waveform viewer



GTKWave - C:\Users\Michele\AppData\Local\lxss\home\lora\exercises\time\lesson_04_srcs\waveform\wave_reset_and_wait.vcd

From: 0 sec To: 42 ns Marker: -- | Cursor: 1250 ps

Type	Signals
wire	CLOCK
wire	NUMBER[31:0]
wire	NUMBER_READY
wire	RESET
wire	RESULT[31:0]
wire	RESULT_READY

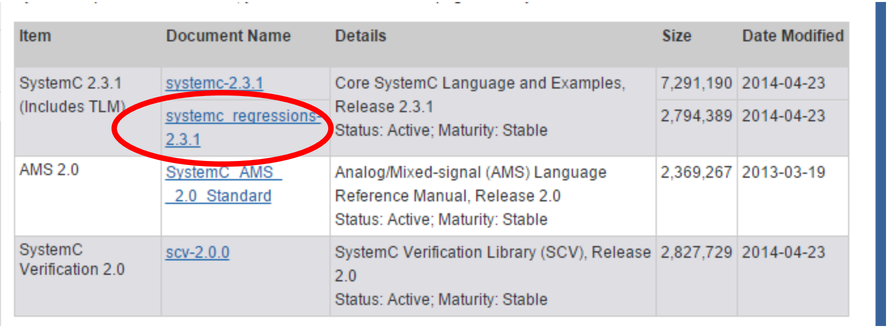
SystemC Into Action

SystemC: material

- C++ references
  - http://www.cplusplus.com/doc/tutorial/
- Elearning website
  - Quick reference
  - Manual
- Download
  - http://www.accellera.org/downloads/standards/systemc

Università degli Studi di Verona

4 Useful references
SystemC design flow
- SystemC manual
- SystemC quick reference
- Makefile manual
- Gdb debugger
- TLM protocol manual



Item	Document Name	Details	Size	Date Modified
SystemC 2.3.1 (Includes TLM)	systemc-2.3.1	Core SystemC Language and Examples, Release 2.3.1 Status: Active; Maturity: Stable	7,291,190	2014-04-23
	systemc_regressions-2.3.1		2,794,389	2014-04-23
AMS 2.0	SystemC_AMS_2.0_Standard	Analog/Mixed-signal (AMS) Language Reference Manual, Release 2.0 Status: Active; Maturity: Stable	2,369,267	2013-03-19
SystemC Verification 2.0	scv-2.0.0	SystemC Verification Library (SCV), Release 2.0 Status: Active; Maturity: Stable	2,827,729	2014-04-23

Installation

- Download SystemC
  - http://www.accellera.org/downloads/standards/systemc
  - From the elearning portal (SystemC 2.3.3 Source Code)
- Unzip the archive

```
$> tar -xzfv systemc-2.3.3.tar.gz
```

- Configure the installation

```
$> cd systemc-2.3.3
$> mkdir build
$> mkdir -p ~/pse_libraries/systemc
$> cd build
$> cmake -DCMAKE_INSTALL_PREFIX=<home_directory>/pse_libraries/systemc ..
```

- Compile

```
$> make -j
$> make install
```

SystemC into action (Cont.d)

- Download the Lesson_01.tar.gz from the e-learning web-site

- Uncompress the archive

```
– $ tar -xzvf 01_systemc.tar.gz
– $ cd 01_systemc/
– $ ls
      • dist div root
```

- Compiling and executing the dist module

```
$ cd dist/
$ ls
      bin  include CMakeLists.txt
$ mkdir build && cd build
$ cmake –DCMAKE_PREFIX_PATH=/home/<your home>/pse_libraries/systemc ..
$ make -j
$ ./dist_RTL
```

SystemC into action (Cont.d)

- Compiling and executing the div module (similar for root module)
  - Same procedure as for the dist module
- The compiler lists a set of errors
  - Source file and the line at which each error was found, and the reason why the code is wrong
  - To find more subtle errors
    - Trace the execution flow by printing on the standard output (e.g. the value of some variables) or by using a debugger (e.g. gdb)
- Have fun ☺

Assignment for this lesson

- Small error report (1-2 pages)
  - Detailing the errors found in the root and the div modules
    - Explain the kind of error (i.e., syntax, unexpected behavior, etc.)
    - The resulting output
    - How to fix them
  - No code needed (solutions will be published next week)
    - **Of course, in the report you must demonstrate to understand whats's wrong and why**