

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

Lecture 3 – The SystemC Language

Sources:

*M. Radetzki, Univ. of Stuttgart
OSCI TLM 2.0 Documentation*

Andreas Gerstlauer

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



The University of Texas at Austin
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 and semantics
 - Channel library
- **SystemC Transaction-Level Modeling (TLM)**
 - SystemC TLM 2.0 library
 - Advanced modeling approaches

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 classes of models
- **Syntax vs. semantics**
 - Syntax defines grammar (textual or graphical)
 - Semantics defines meaning (operational or denotational)
 - Underlying inherent/native execution/simulation model

Programming Languages

- **Imperative programming models**
 - Statements that manipulate program state, control flow
 - Sequential programming languages [C, C++, ...]
 - Van Neuman semantics
- **Declarative programming models**
 - Rules for data manipulation, data flow
 - Functional programming [Haskell, Lisp, Excel]
 - Logic programming [Prolog]
- **Sequential behavior at processor level**
 - Granularity of arithmetic/logic expressions over variables
 - Implicit or explicit operation-level parallelism
 - No coarser-grain concurrency or time

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**
 - Structural concurrency, time
 - Behavioral concurrency at higher levels?

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

5

System-Level Design Languages (SLDLs)

- **Concepts**

	C	C++	Java	VHDL	Verilog	SystemC	Statecharts	SpecCharts	SpecC
Behavioral hierarchy	○	○	○	○	○	○	○	●	●
Structural hierarchy	○	○	○	●	●	●	○	○	●
Concurrency	○	○	◐	●	●	●	●	●	●
Synchronization	○	○	◐	●	●	●	●	●	●
Exception handling	◐	●	●	○	●	○	◐	●	●
Timing	○	○	○	●	●	●	◐	◐	●
State transitions	○	○	○	○	○	○	●	●	●
Composite data types	●	●	●	●	◐	●	○	●	●

○ not supported

◐ partially supported

● supported

Source: R. Doemer, UC Irvine

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

6

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

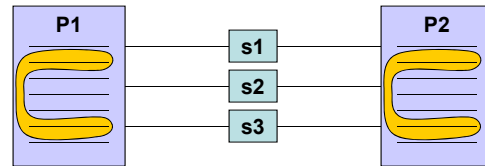
Separation of Concerns

- **Fundamental principle in modeling of systems**
- **Clear separation of concerns**
 - Address separate issues independently
- **System-Level Design Language (SLDL)**
 - Orthogonal concepts
 - Orthogonal constructs
- **System-level Modeling**
 - Computation
 - encapsulated in modules / behaviors
 - Communication
 - encapsulated in channels

Source: R. Doemer, UC Irvine

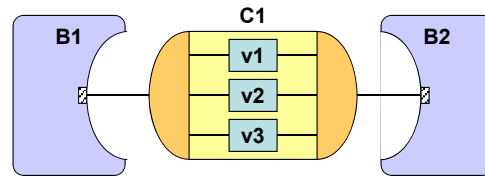
Computation vs. Communication

• Traditional model



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

• SpecC model



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

Source: R. Doemer, UC Irvine

ECE382M.20: SoC Design, Lecture 3

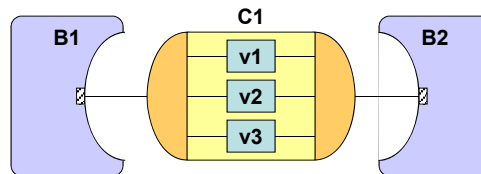
© 2021 A. Gerstlauer

9

Computation vs. Communication

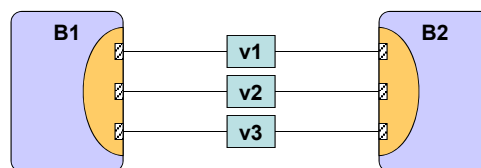
• Protocol Inlining

- Specification model
- Exploration model



- Computation in behaviors
- Communication in channels

• Implementation model



- Channel disappears
- Communication inlined into behaviors
- Wires exposed

Source: R. Doemer, UC Irvine

ECE382M.20: SoC Design, Lecture 3

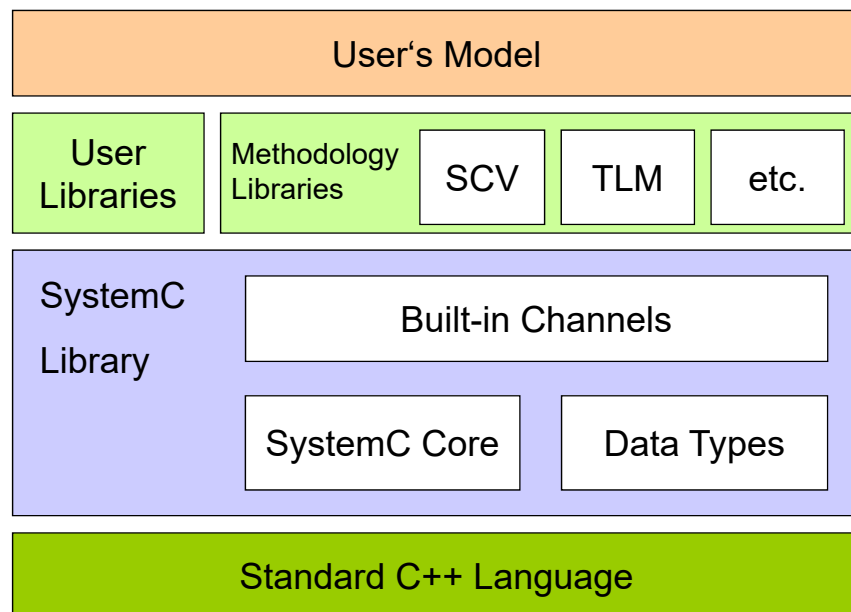
© 2021 A. Gerstlauer

10

Lecture 3: Outline

- ✓ **System design languages**
 - ✓ Goals, requirements
 - ✓ Separation of computation & communication
- **The SystemC language**
 - Basic SystemC model structure
 - Core language syntax and semantics
 - Channel library
- **SystemC Transaction-Level Modeling (TLM)**
 - TLM 2.0 library
 - Advanced modeling approaches

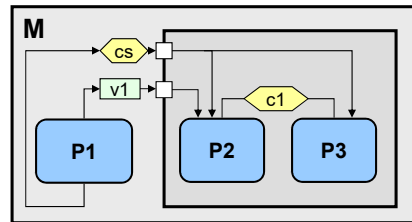
SystemC Class Library Structure



Source: M. Radetzki, Univ. of Stuttgart

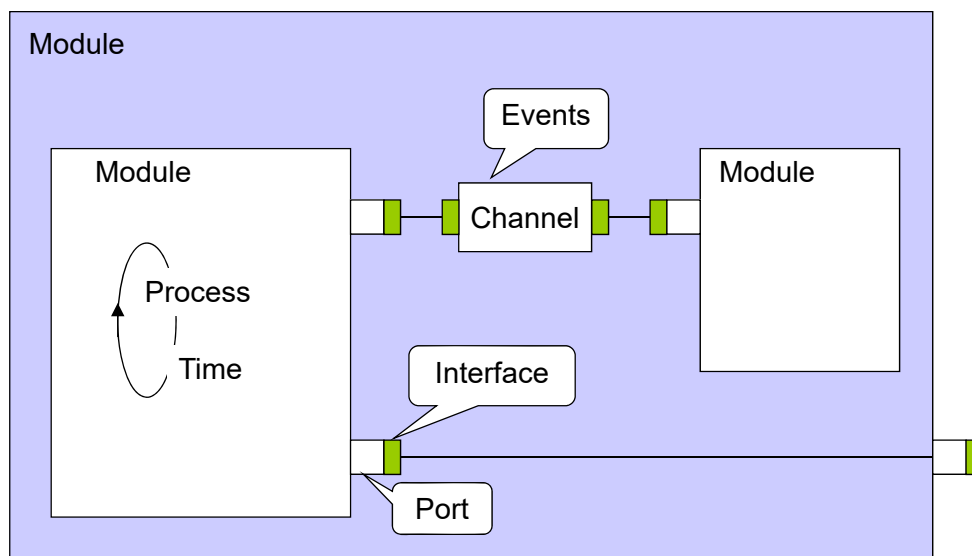
The SystemC Language

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



* SystemC 2.0

SystemC Structure



+ Bit-true data types

Source: M. Radetzki, Univ. of Stuttgart

Modules and Ports

```
// file adder.h
```

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

```
SC_MODULE(Adder)
```

```
{
```

```
  sc_in<int> x;
```

```
  sc_in<int> y;
```

```
  sc_out<int> s;
```

```
  ...
```

```
};
```

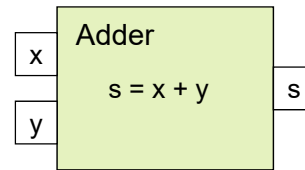
usage of SystemC library

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

© 2021 A. Gerstlauer

15

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;
```

```
  }
```

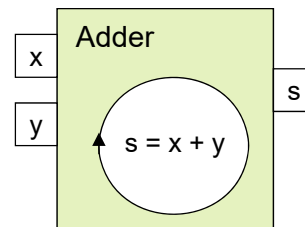
```
};
```

function prototype

module constructor

function registered as a process

activation condition of the process:
new value (value change) on port x
or port y leads to automatic start of add()



Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

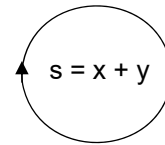
© 2021 A. Gerstlauer

16

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

ECE382M.20: SoC Design, Lecture 3

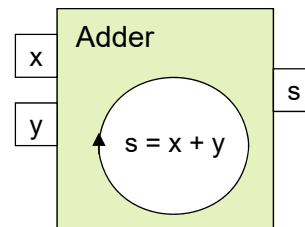
© 2021 A. Gerstlauer

17

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

ECE382M.20: SoC Design, Lecture 3

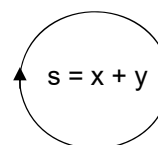
© 2021 A. Gerstlauer

18

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

19

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

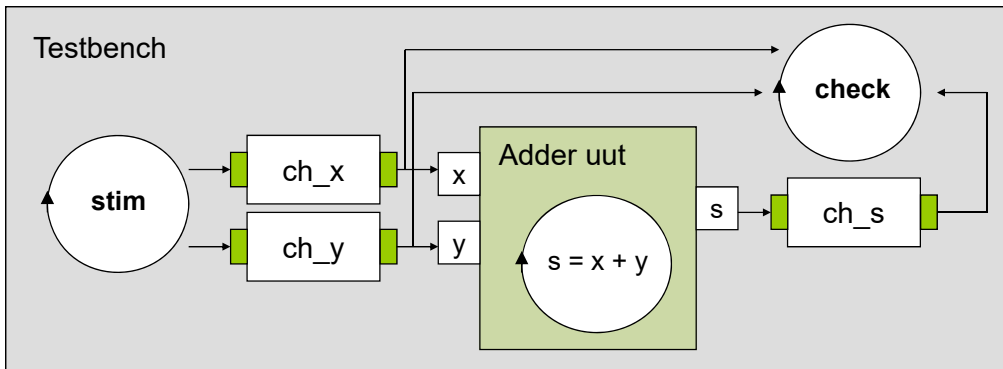
Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

20

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

21

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
        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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

22

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;
}

```

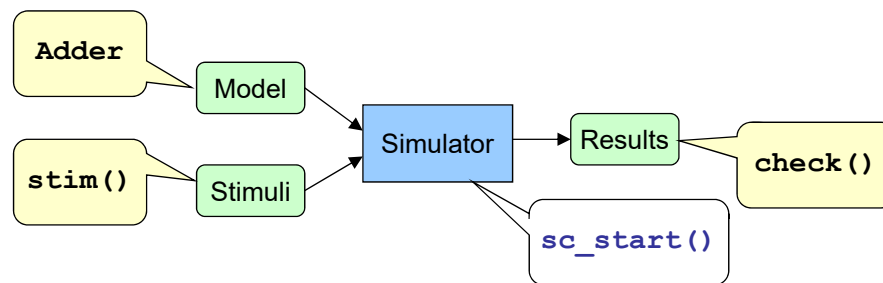
Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

23

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)
- Tracing: `sc_trace` (VCD waveforms, view e.g. with gtkwave)

Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

24

Lecture 3: Outline

- ✓ **System design languages**
 - ✓ Goals, requirements
 - ✓ Separation of computation & communication
- **The SystemC language**
 - ✓ Basic SystemC model structure
 - Core language syntax and semantics
 - Channel library
- **SystemC Transaction-Level Modeling (TLM)**
 - TLM 2.0 library
 - Advanced modeling approaches

Bit Data Types

Type	bool (C++ type)	sc_logic
Values	false (0), true (1)	'0', '1', 'X' (unknown), 'Z' (high-impedance)
Logic Operations	&&, , !, etc. (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

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

Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

27

Arithmetic Data Types

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

Note: can mix sc_int, sc_uint with C++ integer data types

e.g.: sc_int + int is possible

Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

28

Arbitrary Precision Arithmetics

Type	sc_bigint<N> vector of sign and N-1 binary digits	sc_biguint<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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

29

Fixed Point Data Types

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

30

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

31

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

32

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

33

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 << ...`

Dynamic sensitivity

... on a single event

... on a combination of events

all events have happened

at least one event has happened

... for a time period

... timeout (wait on event no longer than *t*)

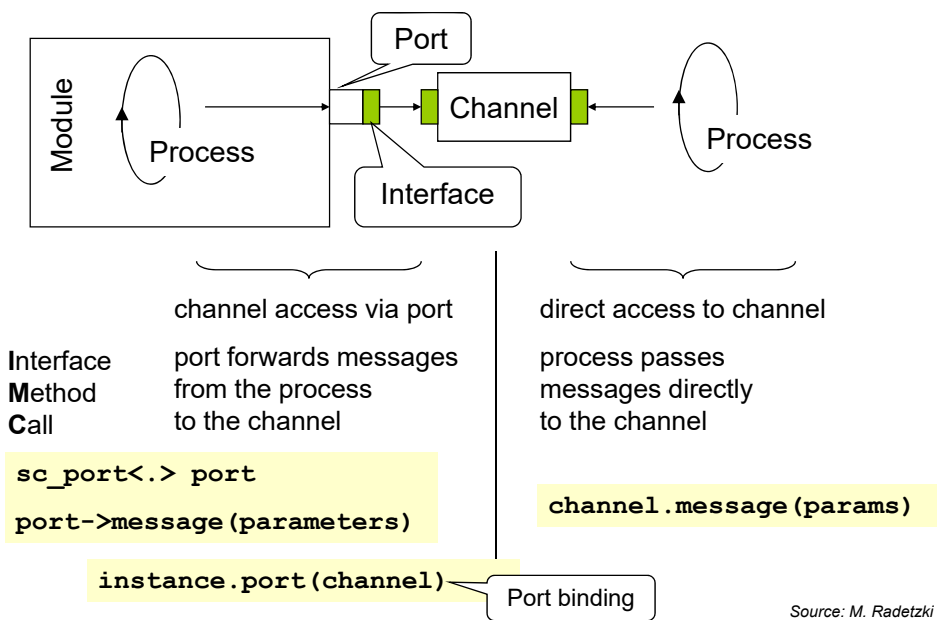
Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

34

SystemC Channels

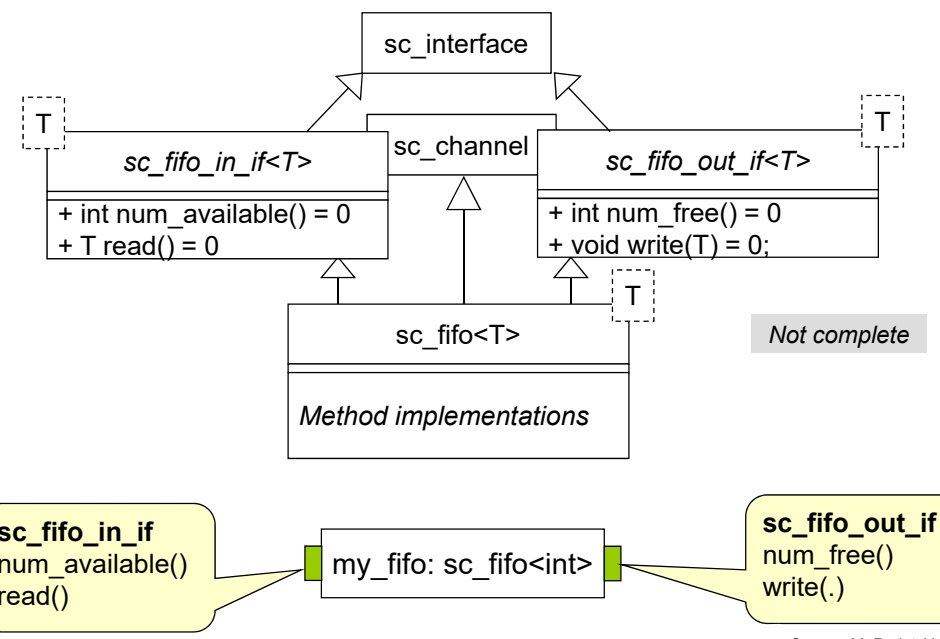


ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

35

Channels & Interfaces (sc_fifo)



ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

36

Putting It All Together

Maximum FIFO size (default: 16)

```

sc_fifo<.> my_fifo(4);
sc_fifo_in_if sc_fifo_in_if;
sc_fifo_out_if sc_fifo_out_if;
    
```

```

m
Module
Process
p
my_fifo
Process
    
```

```

m.p(my_fifo)
my_fifo.write(.)
    
```

```

sc_port<sc_fifo_in_if<.>> p;
Shortcut: sc_fifo_in<.> p;
p->read()
    
```

Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3
© 2021 A. Gerstlauer
37

Exports

port
required interface

no port here ?!

asymmetry

```

m
Module
Process
p
Adapter a
data
req
ack
e
    
```

export
provided IF

```

sc_export<sc_fifo_in_if<T>> e;
    
```

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

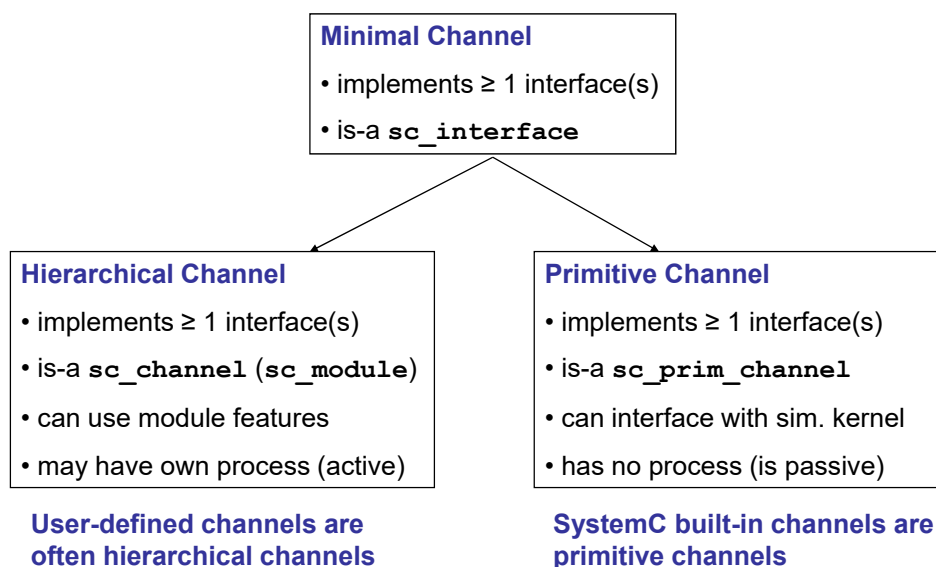
In constructor: `e(*this);` or `e.bind(*this);`

- Note: sc_modules other than adapters may have exports, too.**

Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3
© 2021 A. Gerstlauer
38

SystemC Channels



Source: M. Radetzki

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

39

SystemC Built-In Channels

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

40

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;
};
```

ECE382M.20: SoC Design, Lecture 3

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

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

42

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 topl("Top1");
    sc_start();
    return 0;
}

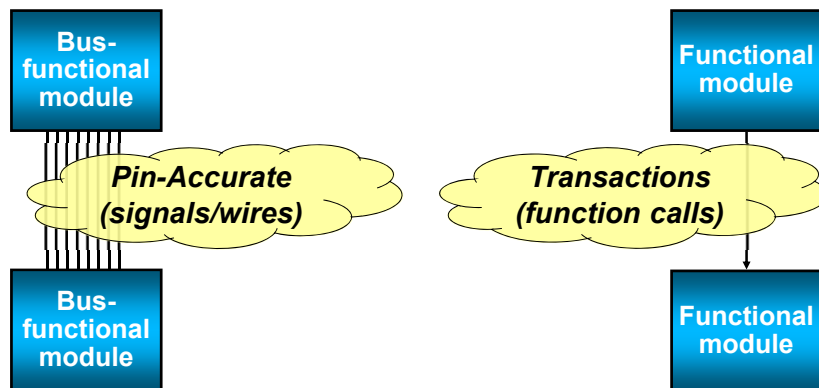
```

Source: S. Swan, Cadence Design Systems

Lecture 3: Outline

- ✓ **System design languages**
 - ✓ Goals, requirements
 - ✓ Separation of computation & communication
- ✓ **The SystemC language**
 - ✓ Basic SystemC model structure
 - ✓ Core language syntax and semantics
 - ✓ Channel library
- **SystemC Transaction-Level Modeling (TLM)**
 - TLM 2.0 library
 - Advanced modeling approaches

Transaction-Level Modeling (TLM)



- **Pin-accurate model (PAM)**
 - Simulate every event (protocols)
- **Transaction-level model (TLM)**
 - Communications by transactions (abstract channels)
 - Granularity of transactions? Dynamic effects?

Source: OSCl TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

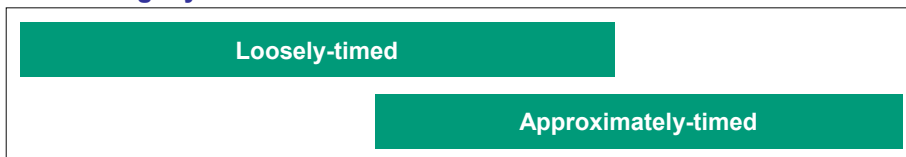
45

SystemC/TLM 2.0

Use cases



TLM-2 Coding styles



Mechanisms



Source: OSCl TLM-2.0

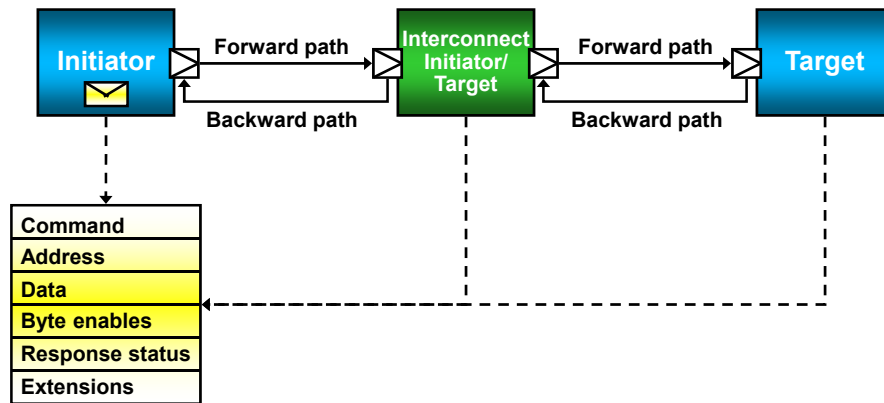
ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

46

Initiators, Targets and Transactions

- Pointer to transaction object is passed from module to module using forward and backward paths
- Transactions are of generic payload type



Source: OSCI TLM-2.0

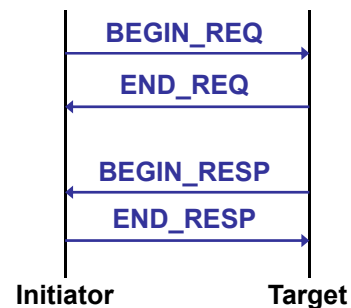
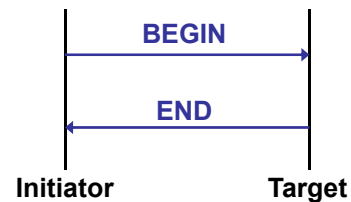
ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

47

SystemC/TLM 2.0 Coding Styles

- **Loosely-timed**
 - Sufficient timing detail to boot OS and simulate multi-core systems
 - Each transaction has 2 timing points: *begin* (call) and *end* (return)
 - One bi-directional call
- **Approximately-timed**
 - Cycle-approximate or cycle-count-accurate
 - Sufficient for architectural exploration
 - Each transaction has at least 4 timing points
 - 4 forward/backward uni-directional calls
 - Must immediately return (no `wait()`)



Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

48

Generic Payload

```

class tlm_generic_payload { Not a template
public:

    // Constructors, memory management
    tlm_generic_payload () ;
    tlm_generic_payload(tlm_mm_interface& mm) ; Construct & set mm
    virtual ~tlm_generic_payload () ; Frees all extensions
    void reset() ; Frees mm'd extensions

    void set_mm(tlm_mm_interface* mm) ; mm is optional
    bool has_mm() ;
    void acquire() ; Incr reference count
    void release() ; Decr reference count,
    int get_ref_count() ; 0 => free trans

    void deep_copy_from(const tlm_generic_payload& other) ;

    ...
};

```

Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

49

Generic Payload Attributes

Attribute	Type	Modifiable?	
Command	tlm_command	No	
Address	uint64	Interconnect only	
Data pointer	unsigned char*	No (array – yes)	Array owned by initiator
Data length	unsigned int	No	
Byte enable pointer	unsigned char*	No (array – yes)	Array owned by initiator
Byte enable length	unsigned int	No	
Streaming width	unsigned int	No	
DMI hint	bool	Yes	Try DMI !
Response status	tlm_response_status	Target only	
Extensions	(tlm_extension_base*) []	Yes	Consider memory management

Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

50

Command, Address and Data

```
enum tlm_command {
    TLM_READ_COMMAND,           Copy from target to data array
    TLM_WRITE_COMMAND,         Copy from data array to target
    TLM_IGNORE_COMMAND         Neither, but may use extensions
};

tlm_command    get_command()    const ;
void           set_command( const tlm_command command ) ;

sc_dt::uint64  get_address()    const;
void           set_address( const sc_dt::uint64 address );

unsigned char* get_data_ptr()   const;           Data array owned by initiator
void           set_data_ptr( unsigned char* data );

unsigned int   get_data_length() const;           Number of bytes in data array
void           set_data_length( const unsigned int length );
```

Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

51

Transport Interfaces

```
template < typename TRANS = tlm_generic_payload >
```

- **Blocking transport interface**

- Typically used with loosely-timed coding style
- `tlm_blocking_transport_if`
`void b_transport(TRANS&, sc_time&);`

- **Non-blocking transport interface**

- Typically used with approximately-timed coding style
- Includes transaction phases
- `tlm_fw_nonblocking_transport_if`
`tlm_sync_enum nb_transport_fw(TRANS&, PHASE&, sc_time&);`
- `tlm_bw_nonblocking_transport_if`
`tlm_sync_enum nb_transport_bw(TRANS&, PHASE&, sc_time&);`

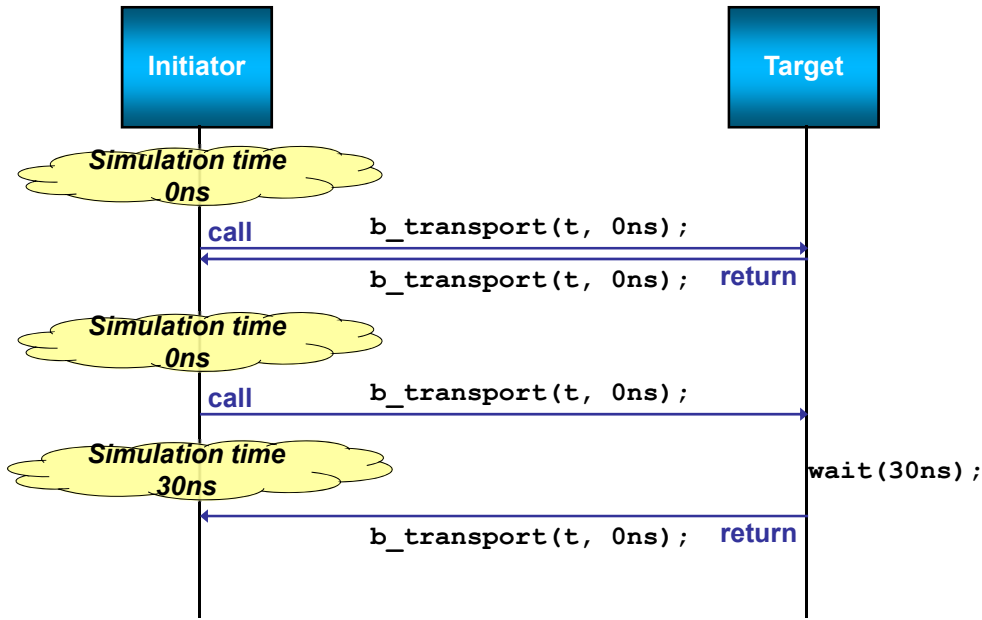
Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

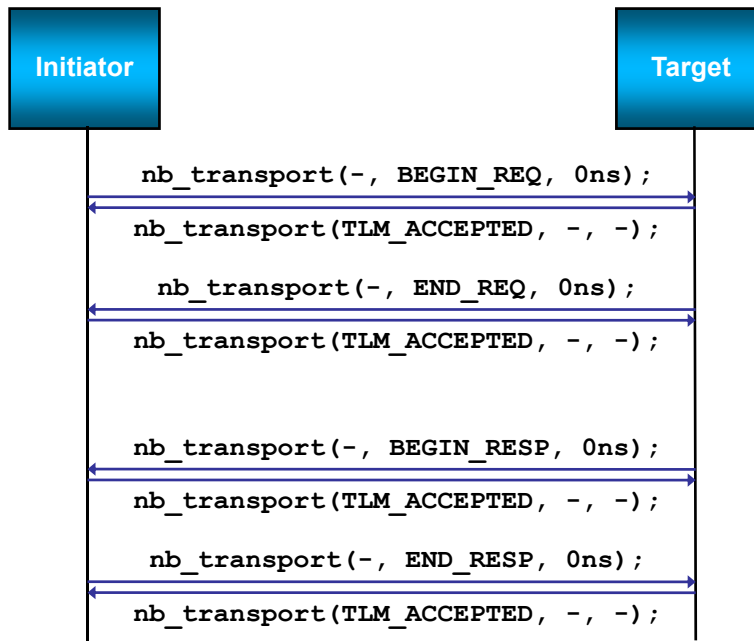
52

Loosely Timed (LT) Model



Source: OSCI TLM-2.0

Approximately Timed (AT) Model



Source: OSCI TLM-2.0

NB Return Values (tlm_sync_enum)

- **TLM_ACCEPTED**
 - Transaction, phase and timing arguments unmodified (ignored) on return
 - Target may respond later (depending on protocol)
- **TLM_UPDATED**
 - Transaction, phase and timing arguments updated (used) on return
 - Target has advanced the protocol state machine to the next state
- **TLM_COMPLETED**
 - Transaction, phase and timing arguments updated (used) on return
 - Target has advanced the protocol state machine straight to the final phase

Source: OSCI TLM-2.0

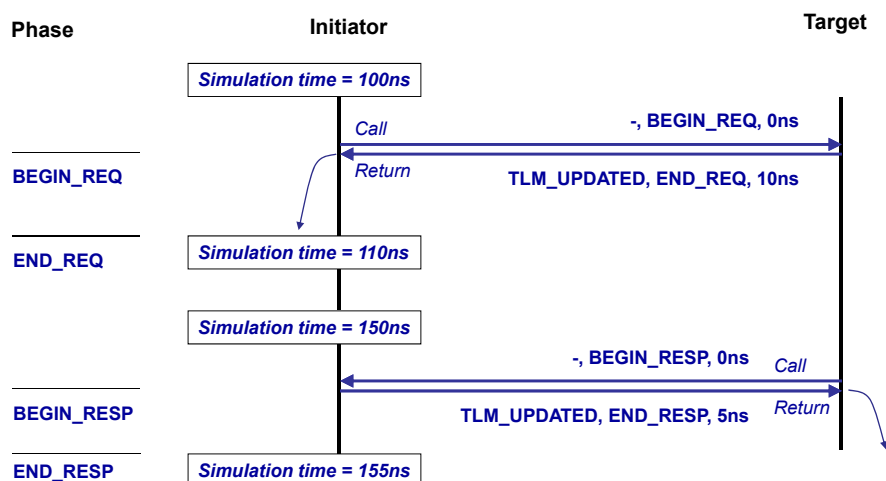
ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

55

Using the Return Path

- Callee annotates delay to next transaction
 - Caller waits



Source: OSCI TLM-2.0

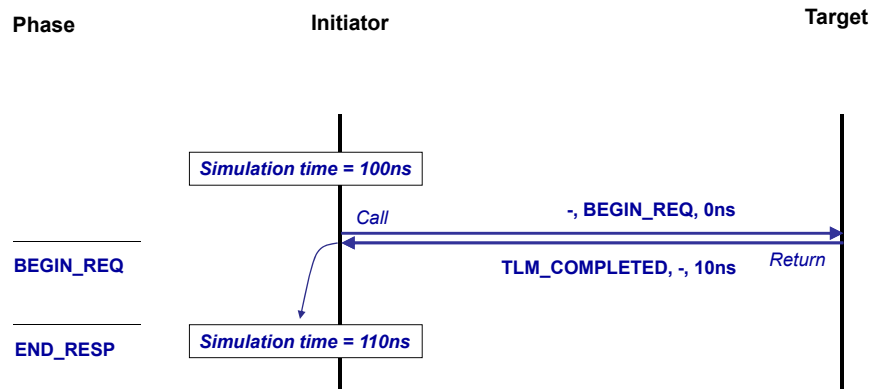
ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

56

Early Completion

- Callee annotates delay to next transaction
 - Caller waits



Source: OSC TLM-2.0

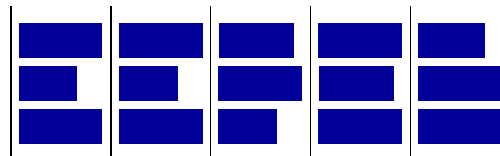
ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

57

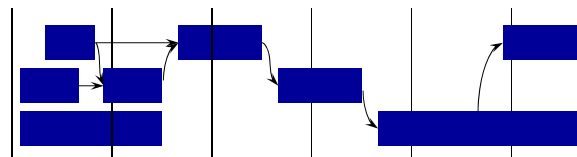
Temporal Decoupling

- Loose coupling
 - OS and driver SW development
 - Local process time
 - Every process runs ahead until data is missing or a time quantum boundary was reached (local/global time synchronization)



- Approximate coupling

- Architecture trade-off
- Each process has the global SystemC time, processes synchronize
- Time may be accurate or estimated



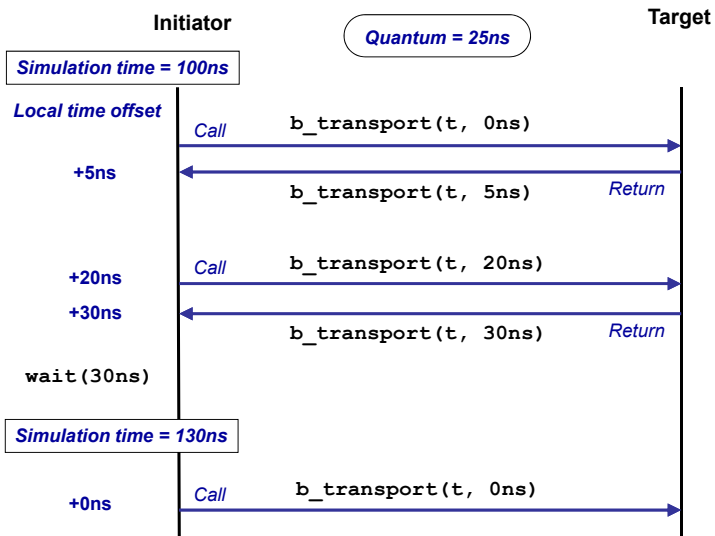
Source: W. Ecker, Infineon

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

58

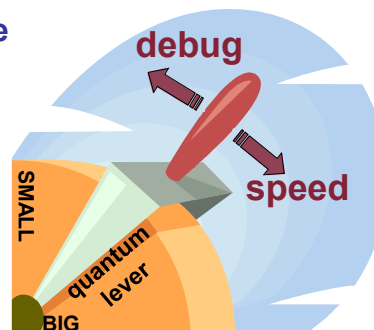
Time Quantum



Source: OSCI TLM-2.0

The Quantum Keeper (tlm_quantumkeeper)

- Quantum is user-configurable

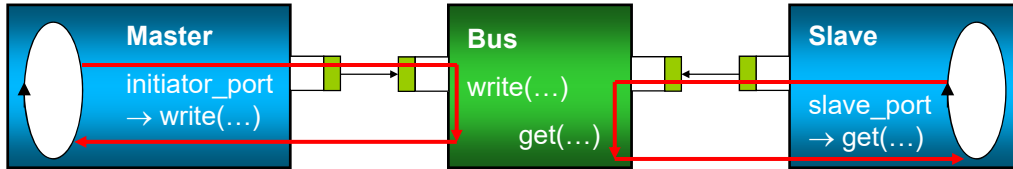


- Processes can check local time against quantum

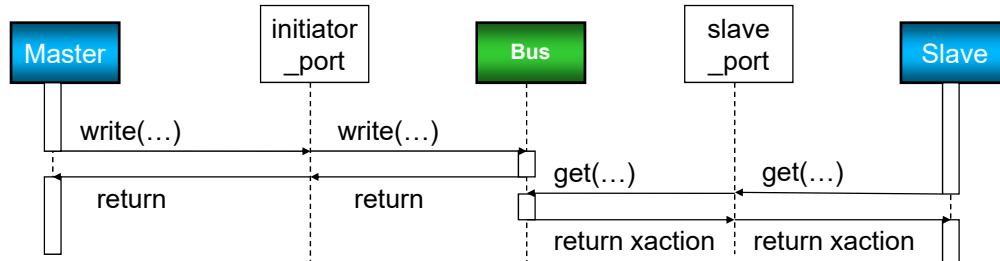
Source: OSCI TLM-2.0

Active Slaves

Slaves have own thread, actively get bus transactions

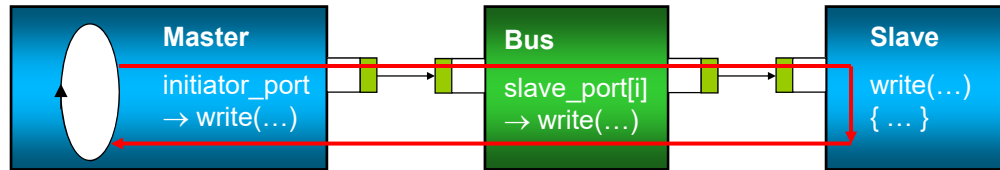


Blocking or non-blocking transport mechanism
 May model an active, non-blocking bus with transaction delays

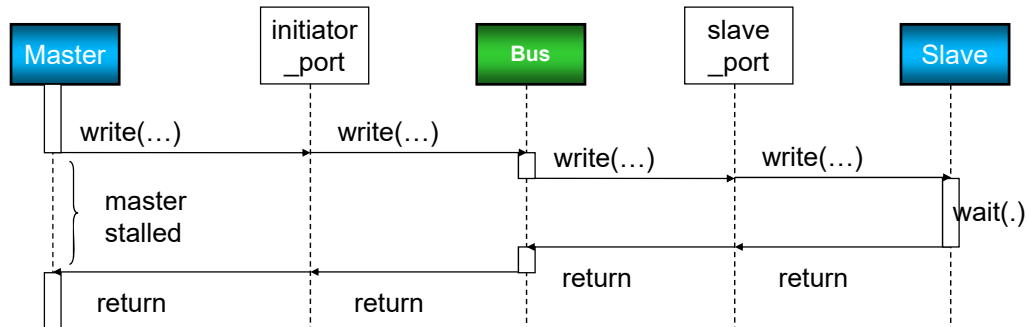


Passive Slaves

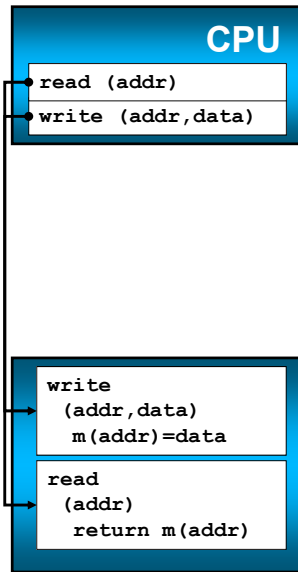
Slave methods executed by the master's thread.



The master is blocked while bus / slave uses its thread.
 No context switch overhead



Direct Memory Interface (DMI)



- **Implementation approach**
 - Direct member function call of target object
 - In master context
 - Pure virtual interface classes defined interface
 - Function call encapsulates and hides all interconnect details
 - Port/export provides connection semantics (incl. restrictions)
- **Direct memory interface**
 - Faster (no context switch)
- **Debug interface**

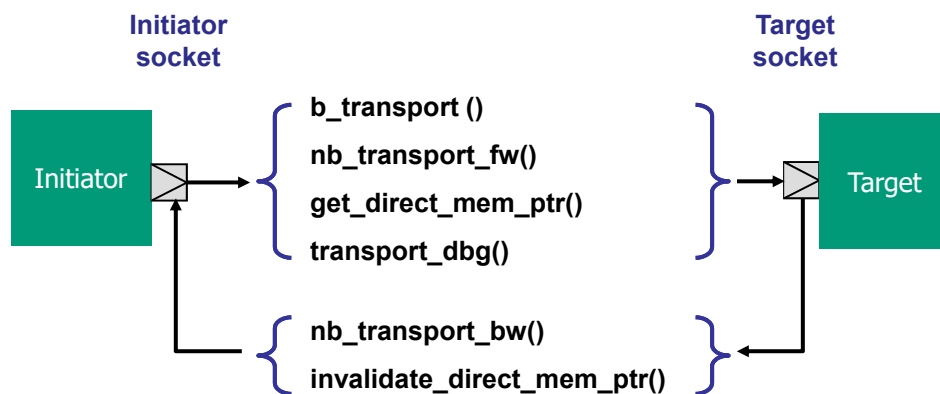
Source: W. Ecker, Infineon

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

63

Sockets



- **Sockets combine fw and bw paths and group interfaces**

Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

64

Convenience Sockets

- The “simple” sockets
 - `simple_initiator_socket` and `simple_target_socket`
 - In namespace `tlm_utils`
 - Derived from base sockets
`tlm_initiator_socket` and `tlm_target_socket`
- “simple” because they are simple to use
 - Do not bind sockets (ports) to objects (implementations)
 - Instead, register methods with each socket
 - Do not allow hierarchical binding
- Not obliged to register both `b_transport` and `nb_transport`
 - Automatic conversion (assumes base protocol)
 - Variant with no conversion – `passthrough_target_socket`

Source: OSCl TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

65

Simple Socket Example (Initiator)

```
class Initiator: public sc_module
{
    tlm_utils::simple_initiator_socket<Initiator> init_socket;

    SC_CTOR(Initiator): init_socket("init_socket")
    {
        // Register methods for backward path
        init_socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
        init_socket.register_invalidate_direct_mem_ptr(this,
            &Initiator::invalidate_direct_mem_ptr);

        SC_THREAD(thread);
    }

    void thread() { ...
        // Call methods on forward path
        init_socket->b_transport( trans, delay );
        init_socket->nb_transport_fw( trans, phase, delay );
        init_socket->get_direct_mem_ptr( trans, dmi_data );
        init_socket->transport_dbg( trans );
    }

    // Methods on backward path
    virtual tlm::tlm_sync_enum nb_transport_bw( ... );
    virtual void invalidate_direct_mem_ptr( ... );
};
```

Source: OSCl TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

66

Simple Socket Example (Target)

```
class Target: public sc_module
{
    tlm_utils::simple_target_socket<Target> targ_socket;

    SC_CTOR(Target): targ_socket("targ_socket")
    {
        // Register methods for forward path
        targ_socket.register_nb_transport_fw( this, &Target::nb_transport_fw);
        targ_socket.register_b_transport(    this, &Target::b_transport);
        targ_socket.register_transport_dbg(  this, &Target::transport_dbg);
        targ_socket.register_get_direct_mem_ptr(this,
                                                &Target::get_direct_mem_ptr);

        SC_THREAD(thread);
    }

    void thread() { ...
        // Call methods on backward path
        targ_socket->nb_transport_bw( ... );
        targ_socket->invalidate_direct_mem_ptr( ... );
    }

    // Methods on forward path
    virtual void b_transport( ... );
    virtual tlm::tlm_sync_enum nb_transport_fw( ... );
    virtual bool get_direct_mem_ptr( ... );
    virtual unsigned int transport_dbg( ... );
};
```

Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

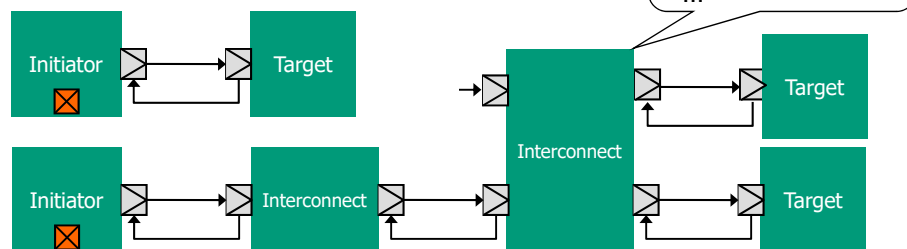
67

Simple Socket Example (Top)

- Bind initiator to target sockets

```
SC_MODULE(Top) {
    Initiator *init;
    Target *targ;
    SC_CTOR(Top) {
        init = new Initiator("init");
        targ = new Target("targ");
        init->init_socket.bind( targ->targ_socket );
    }
};
```

- Define system-level connectivity



Source: OSCI TLM-2.0

ECE382M.20: SoC Design, Lecture 3

© 2021 A. Gerstlauer

68

Lecture 3: Summary

- **SystemC base language**
 - C++ class library
 - Don't invent a new language, leverage existing tools
 - De-facto industry-standard
 - SoC architecture & virtual platform modeling
- **SystemC Transaction-Level Modeling (TLM)**
 - TLM 2.0 class library on top of base language
 - Evolution from TLM 1.0
 - Detailed documentation as part of TLM install
 - `/usr/local/packages/systemc-2.3.1/docs/tlm/release`
 - Speed vs. accuracy in bus/transaction modeling
 - Various TLM modeling approaches in literature & research
 - Modeling of arbitration, bus pipelining, split transactions, ...