Embedded Systems

Thursday

- Midterm exam: December 16th, 2010, AudiMO, 16:00 - 19:00

- No lecture on December 16th

- Open book: bring any handwritten or printed notes, or any books you like.
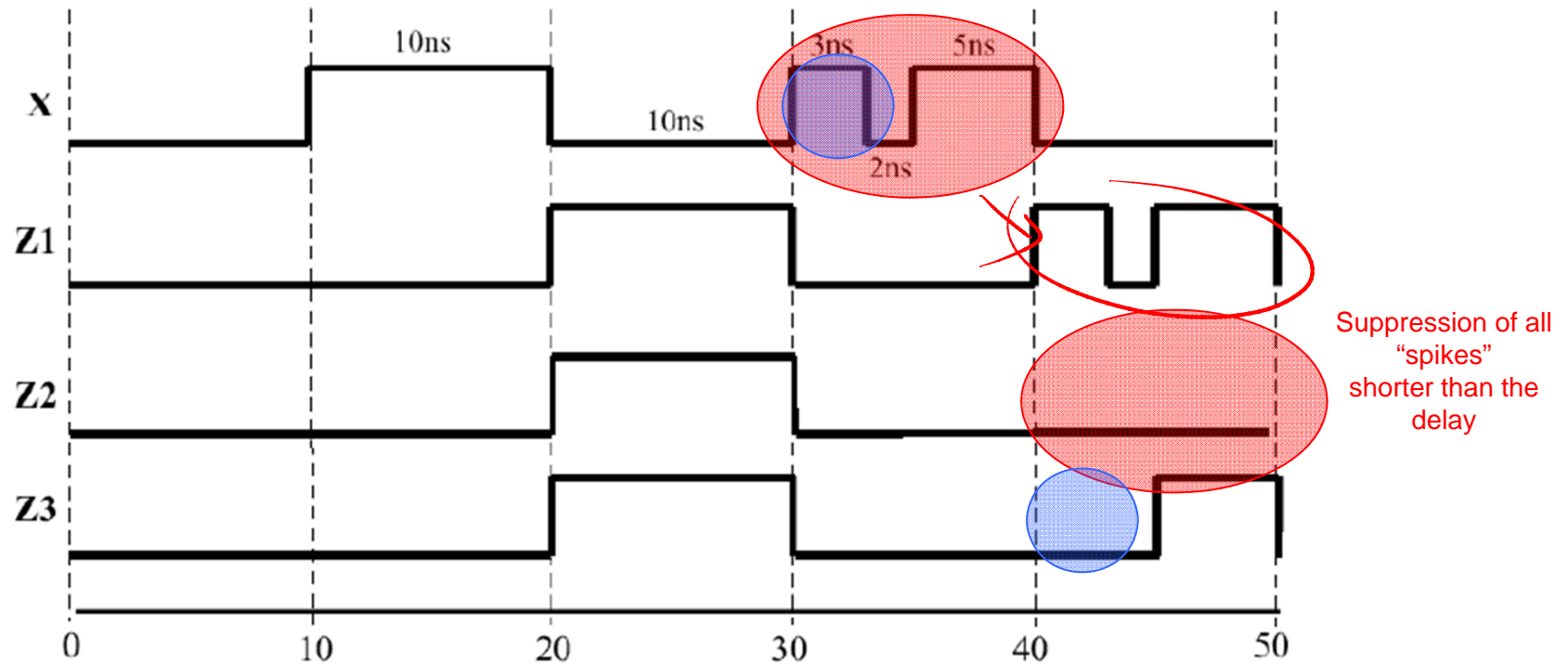
- Please bring your ID.

Exam Policy

- Midterm/End-of-Term Exam/End-of-Semester Exam

Requirement for admission to end-of-term and end-of-semester exams:

- > 50% of points in problem sets,
- > 50% of points in each project milestone, and
- > 50% of points in midterm exam

- **Final grade:**
- best grade in end-of-term or end-of-semester exam

Example

Suppression of all "spikes" shorter than the delay

shorter than the indicated suppression threshold

Z1 <= **transport** X **after** 10 ns; -- transport delay
Z2 <= X **after** 10 ns; -- inertial delay
Z3 <= **reject** 4 ns X **after** 10 ns; -- delay with specified rejection pulse width

CS - ES

- 4 -

Modeling Behavior

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral* architecture
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
    - *sequential statements*, including
    - *signal assignment statements* and
    - *wait statements*

Modeling Structure

- *Structural* architecture
    - implements the module as a composition of subsystems
    - contains
        - *signal declarations*, for internal interconnections
            - the entity ports are also treated as signals
        - *component instances*
            - instances of previously declared entity/architecture pairs
        - *port maps* in component instances
            - connect signals to component ports
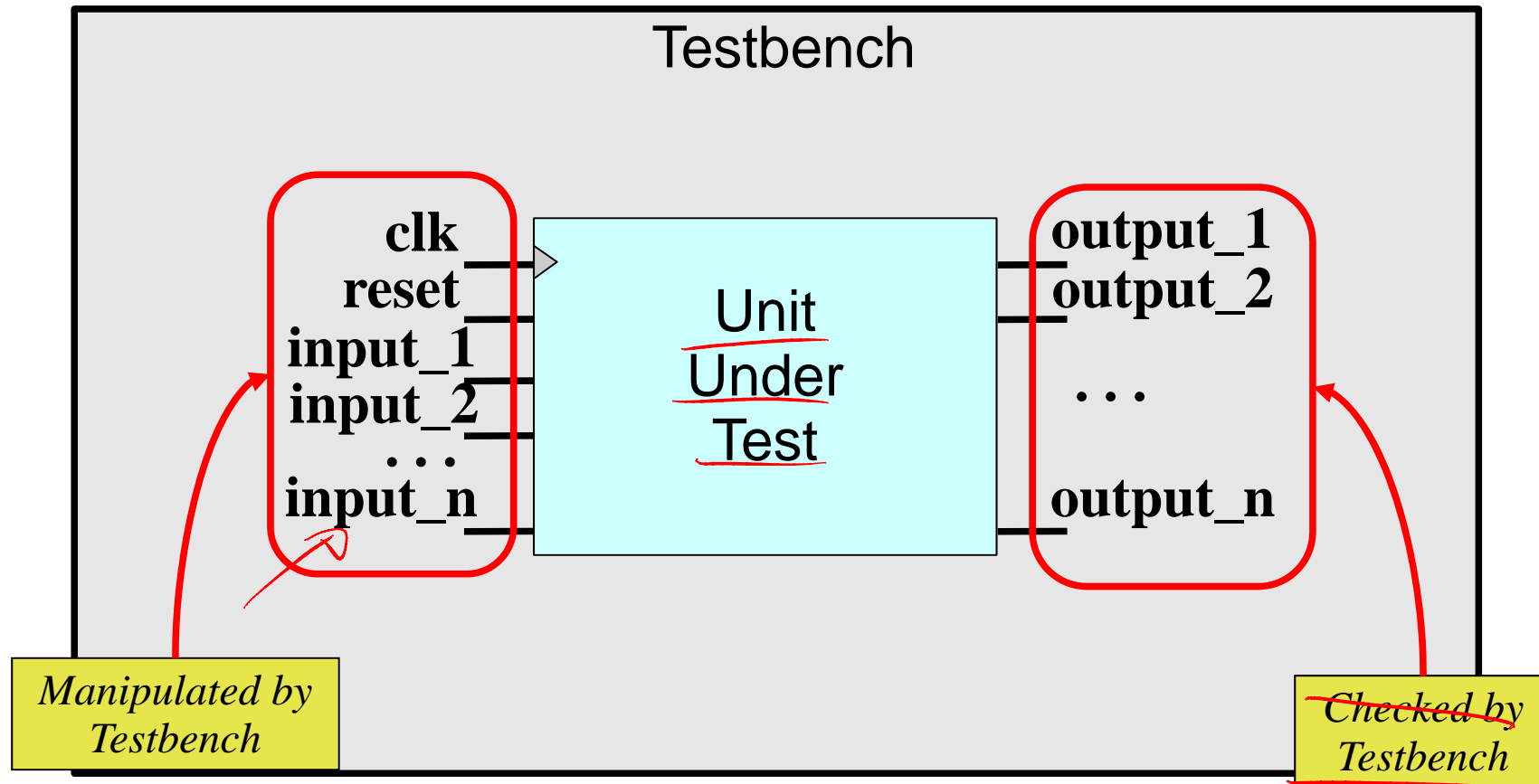   - *wait statements*

Structure Example
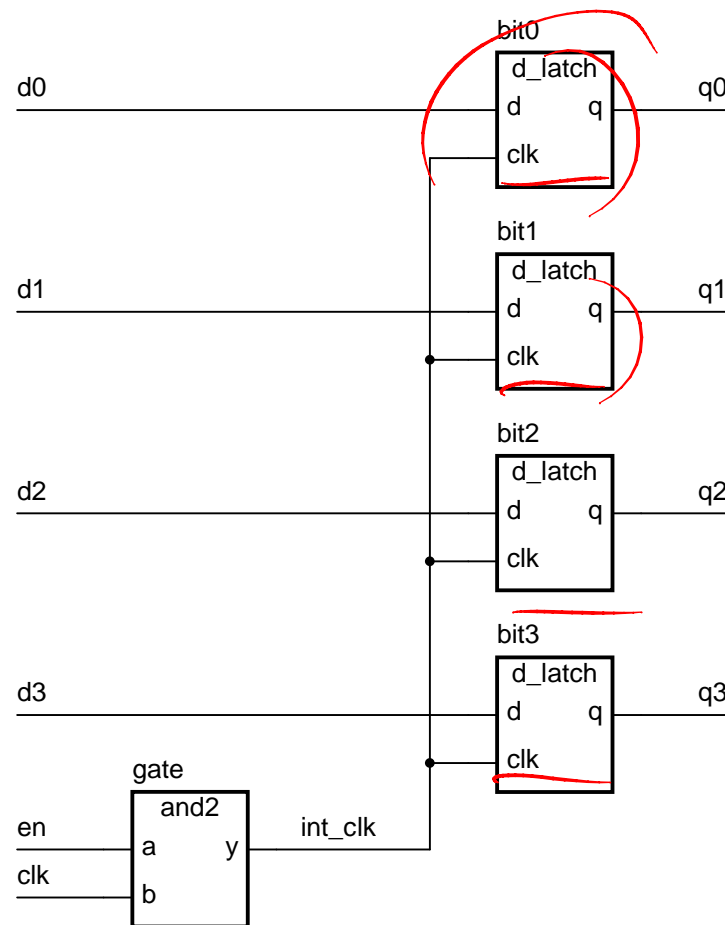

```
architecture struct of reg4 is
    signal int_clk : bit;
begin
    bit0 : entity work.d_latch(basic)
        port map ( d0, int_clk, q0 );
    bit1 : entity work.d_latch(basic)
        port map ( d1, int_clk, q1 );
    bit2 : entity work.d_latch(basic)
        port map ( d2, int_clk, q2 );
    bit3 : entity work.d_latch(basic)
        port map ( d3, int_clk, q3 );
    gate : entity work.and2(basic)
        port map ( en, clk, int_clk );
end architecture struct;
```
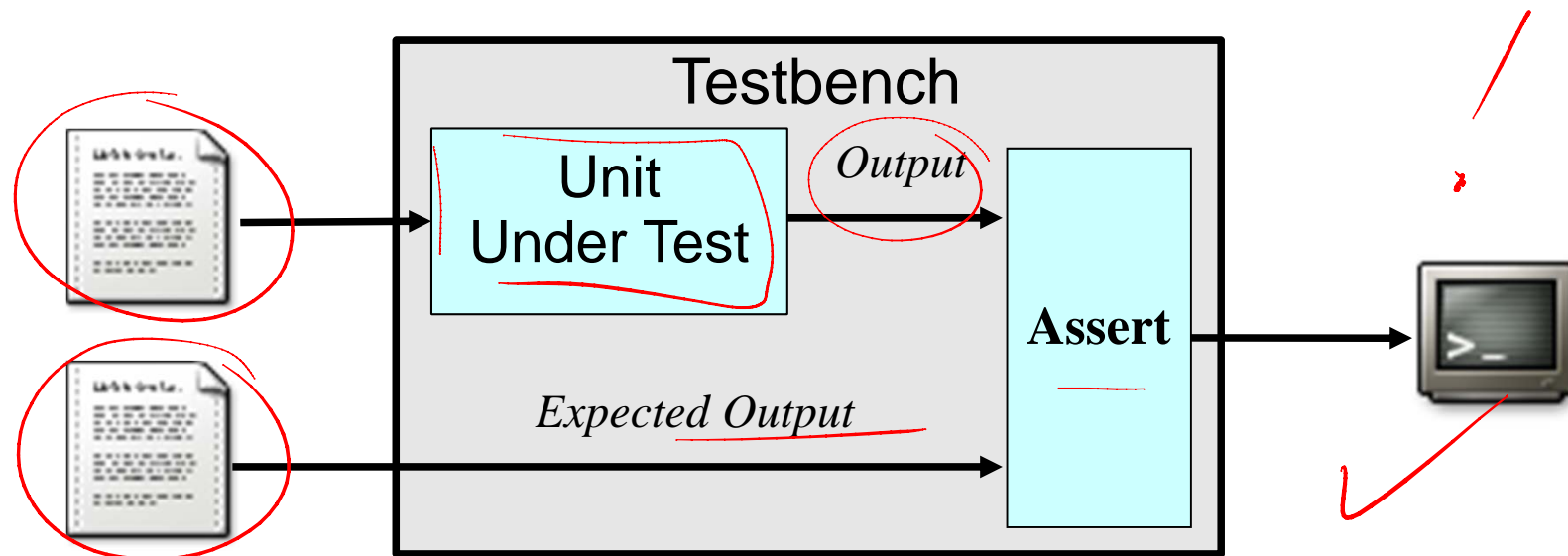
CS - ES

Testbench Structure

Testbench

Unit
Under
Test

clk
reset
input_1
input_2
. . .
input_n

output_1
output_2

. . .

output_n

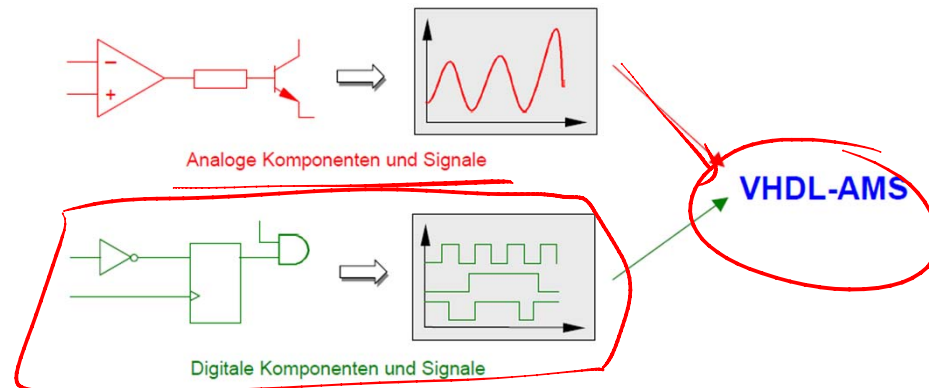*Manipulated by Testbench*

Checked by Testbench

File Input

- Using <u>assert</u> to check behaviour implies writing correct behaviour twice

- Instead, write expected behaviour in file:

Introduction to VHDL-AMS

Introduction

Analoge Komponenten und Signale

Digitale Komponenten und Signale

VHDL-AMS

- **VHDL 1076 is suitable for modeling and simulating discrete systems**
- **Many of today's designs include at least some continuous characteristics:**
    - **System design**
        - **Mixed-signal electrical designs**
        - **Mixed electrical/non-electrical designs**
    - **Analog design**
        - **Analog behavioral modeling and simulation**
    - **Digital design**
        - **Detailed modeling (e.g. submicron effects)**
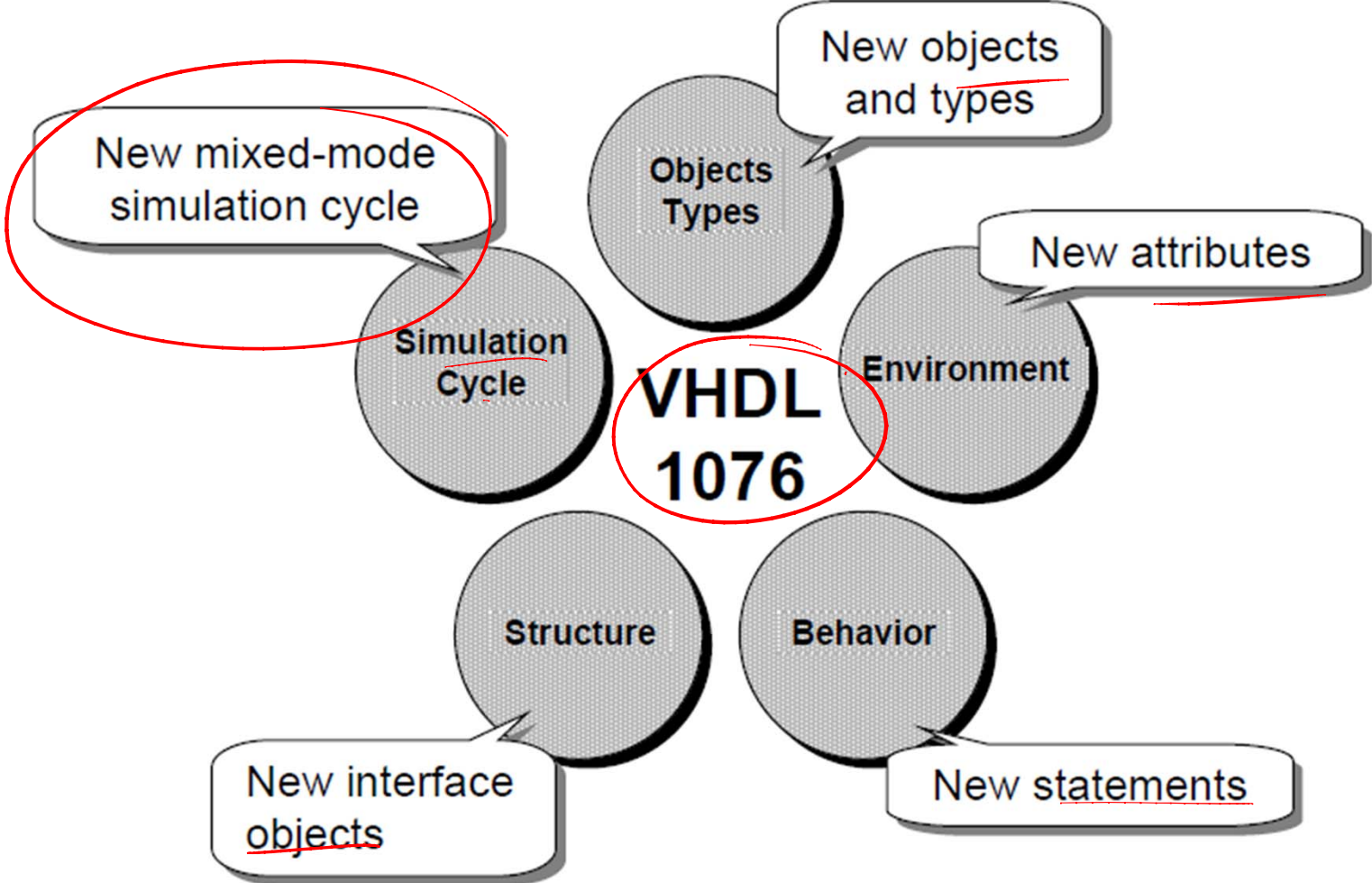- **Designers want a uniform description language**

The world isn‘t digital
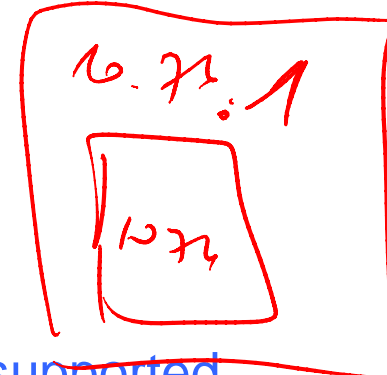
Mixed-Signal-IC
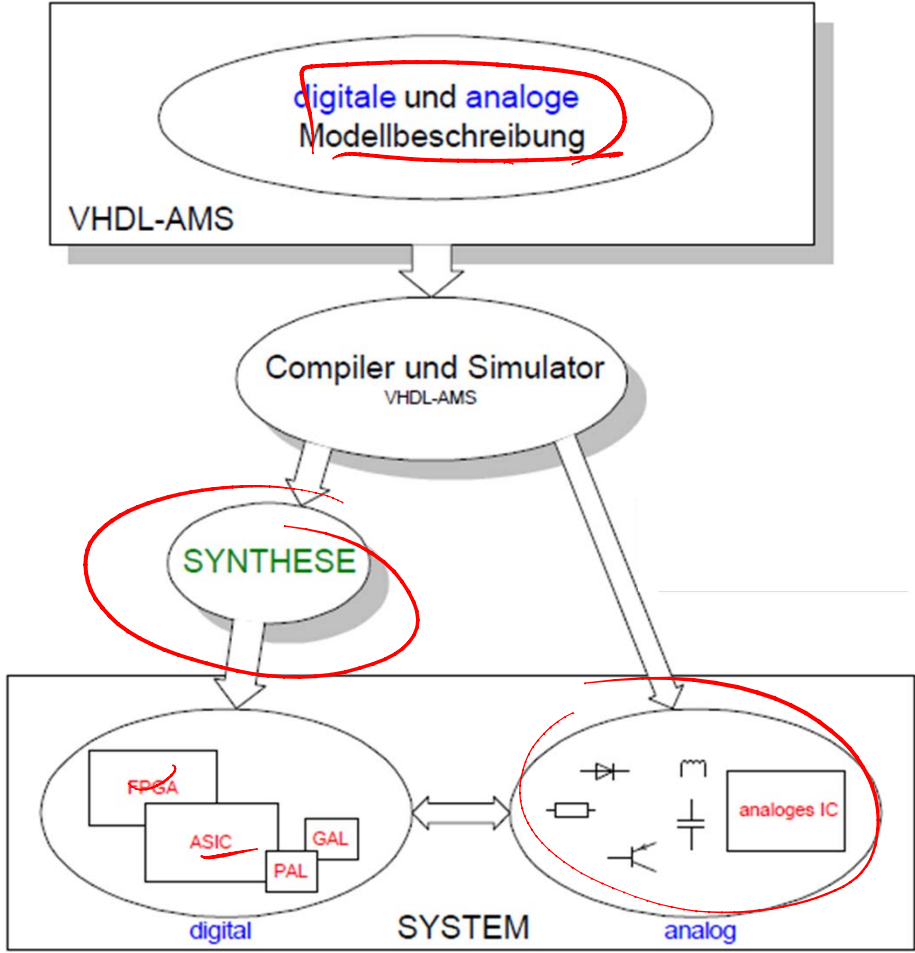
CS - ES
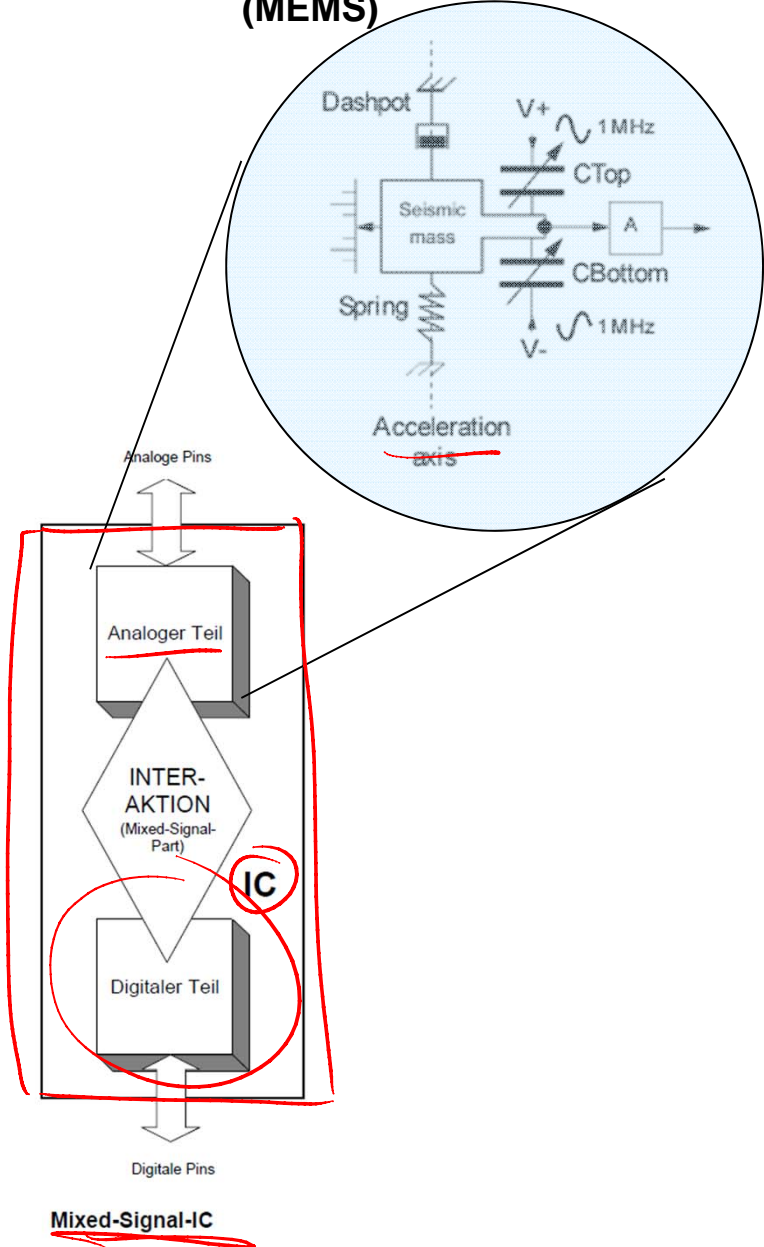
VHDL-AMS Language Architecture

VHDL-AMS Highlights (1)

- **Superset of VHDL 1076**
  - Full VHDL 1076 syntax and semantics is supported

- **Adds new simulation model supporting continuous behavior**
  - Continuous models based on differential algebraic equations (DAEs)

  - DAEs solved by dedicated simulation kernel: the analog solver

  - Handling of initial conditions, piecewise-defined behavior, and discontinuities

VHDL-AMS Highlights (2)

- Extended structural semantics
    - Conservative semantics to model physical systems
        - e.g. Kirchhoff's laws for electrical circuits
    - Non-conservative semantics for abstract models
        - Signal-flow descriptions
    - Mixed-signal interfaces
        - Models can have digital and analog ports
- Mixed-signal semantics
    - Unified model of time for a consistent synchronization of
        - mixed event-driven/continuous behavior
    - ·Mixed-signal initialization and simulation cycle
    - ·Mixed-signal descriptions of behavior
- Frequency domain support
    - Small-signal frequency and noise modeling and simulation

Multiple Architectures per Entity
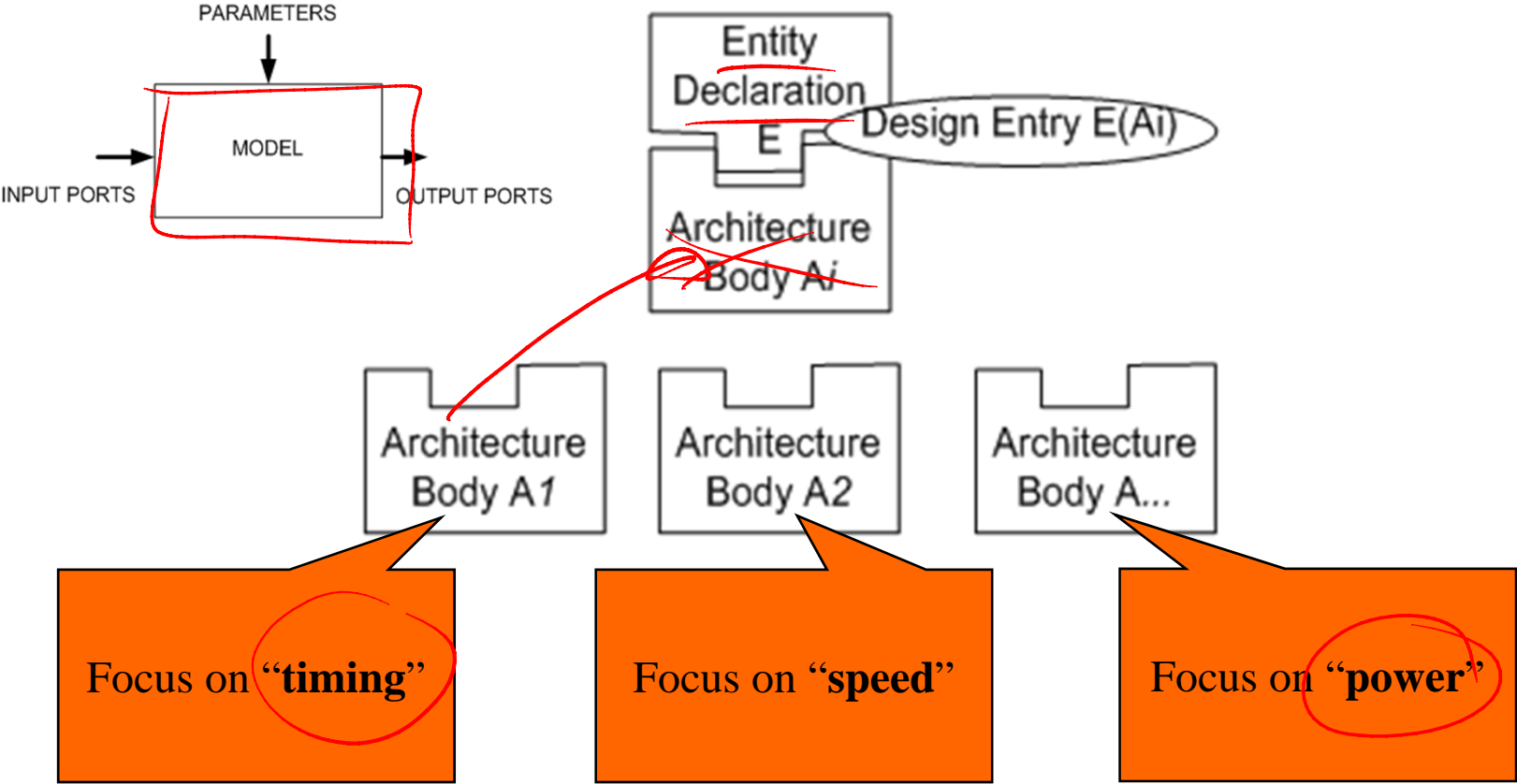
First approximation: one-dimensional spring model
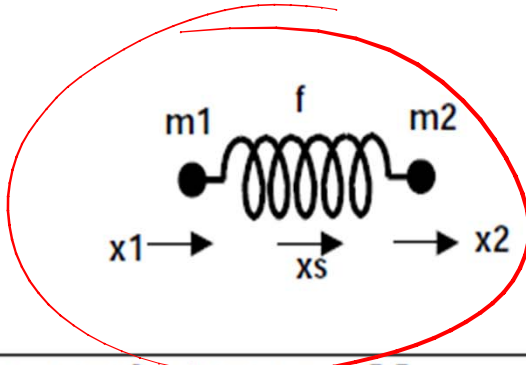
$$m_1 \ddot{x}_1 = - f \bullet (x_1 - x_2)$$

$$m_2 \ddot{x}_2 = - f \bullet (x_2 - x_1)$$

$$x_s = (m_1 x_1 + m_2 x_2)/(m_1 + m_2)$$

$$Energy = 0.5 \bullet (m_1 \dot{x}_1^2 + m_2 \dot{x}_2^2 + f \bullet (x_1 - x_2)^2)$$

```
use work.types.all;
entity Vibration is
end entity Vibration;

architecture H2 of Vibration is    -- hydrogen molecule
  quantity x1, x2, xs: displacement;
  quantity energy: REAL;
  constant m1, m2: REAL := 1.00794*1.6605655e-24;
  constant f: REAL := 496183.3;
begin
  x1'dot'dot == -f*(x1 - x2) / m1;
  x2'dot'dot == -f*(x2 - x1) / m2;
  xs == (m1*x1 + m2*x2)/(m1 + m2);
  energy == 0.5*(m1*x1'dot**2 + m2*x2'dot**2 + f*(x1-x2)**2);
end architecture H2;
```

Quantities
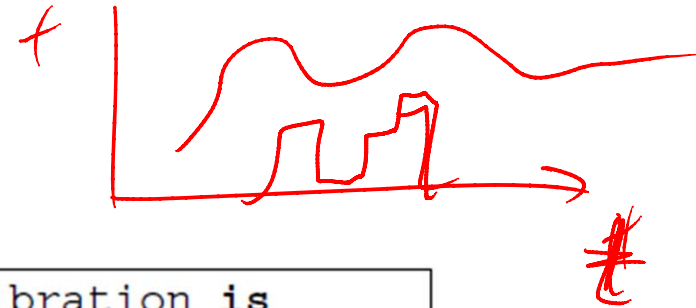
```
architecture H2 of Vibration is

    ...
    quantity x1, x2, xs: displacement;
    quantity energy: REAL;

    ...
begin

    ...
```

- New object in VHDL 1076.1
- Represents an unknown in the set of differential algebraic equations implied by the text of a model
- Continuous-time waveform
- Scalar subelements must be of a floating-point type
- Default initial value for scalar subelements is 0.0

Simultaneous Statements (1)

```
architecture H2 of Vibration is
    ...
begin
    x1'dot'dot == -f*(x1 - x2) / m1;
    x2'dot'dot == -f*(x2 - x1) / m2;
    xs == (m1*x1 + m2*x2)/(m1 + m2);
    energy == 0.5*(m1*x1'dot**2 +
              m2*x2'dot**2 + f*(x1-x2)**2);
end architecture H2;
```

- New class of statements in VHDL 1076.1
- Simple simultaneous statements express relationships between quantities
  - Left-hand side and right-hand side must be expressions with scalar subelements of a floating point type
  - Statement is symmetrical w.r.t. its left-hand and right-hand sides
  - Expressions may involve quantities, constants, literals, signals, and (possibly user-defined) functions
  - At least one quantity must appear in a simultaneous statement

CS - ES

Simultaneous Statements (2)

- Analog solver is responsible for computing the values of the quantities such that the relationships hold (subject to tolerances)

- Simultaneous statements may appear anywhere a concurrent statements may appear

- The order of simultaneous statements does not matter

- Other forms for simultaneous statements:
    - Simultaneous if statement
    - Simultaneous case statement
    - Simultaneous procedural statement

Tolerances (1)

- Numerical algorithms used by analog solver can only find an approximation of the exact solution

- Tolerances are used to specify how good the solution must be

- Each quantity and each simultaneous statement belongs to a tolerance group indicated by a string expression

  - All members of a tolerance group have the same tolerance characteristics

- The language does not define how a tool uses tolerance groups !!

Tolerances (2)
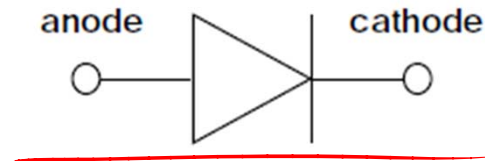
- A quantity gets its tolerance group from its subtype

```
package types is
   subtype displacement is REAL
                            tolerance "default_displacement";
   ...
end package types;
```

```
architecture H2 of Vibration is   -- hydrogen molecule
   quantity x1, x2, xs: displacement;
   quantity energy: REAL;
   ...
```
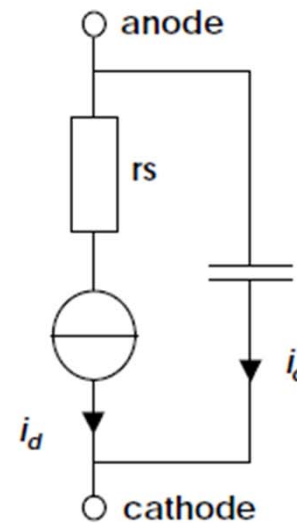
Parameterized Diode

- Example of a conservative model of an electrical component

- Simple large-signal model

$$i_d = \text{is} \cdot \left(e^{(v_d - \text{rs} \cdot i_d)/\text{n} \cdot \text{vt}} - 1 \right)$$

$$i_c = \frac{d}{dt} \left(\text{tt} \cdot i_d - 2 \cdot \text{cj0} \cdot \sqrt{\text{vj}^2 - \text{vj} \cdot v_d} \right)$$

VHDL-AMS Model of Diode

```
library IEEE, Disciplines;
use Disciplines.electrical_system.all;
use IEEE.math_real.all;
entity Diode is
    generic (iss: REAL := 1.0e-14;
             n, af: REAL := 1.0;
             tt, cj0, vj, rs, kf: REAL := 0.0);
    port (terminal anode, cathode: electrical);
end entity Diode;

architecture Level0 of Diode is
    quantity vd across id, ic through anode to cathode;
    quantity qc: charge;
    constant vt: REAL := 0.0258;      -- thermal voltage
begin
    id == iss * (exp((vd-rs*id)/(n*vt)) - 1.0);
    qc == tt*id - 2.0*cj0 * sqrt(vj**2 - vj*vd);
    ic == qc'dot;
end architecture Level0;
```

Terminal
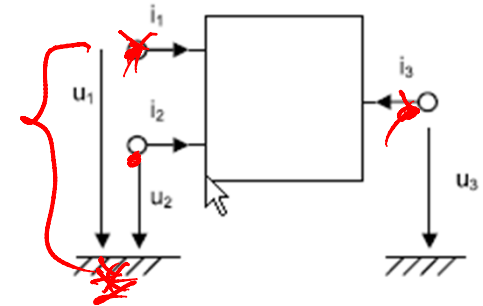

```
library IEEE, Disciplines;
use Disciplines.electrical_system.all;
...
entity Diode is
   ...
   port (terminal anode, cathode: electrical);
end entity Diode;
```

- New object in VHDL 1076.1
- Basic support for structural composition with conservative semantics
- Belongs to a nature
  - Nature electrical defined in package *electrical_system*

Nature

- Represents a physical discipline or energy domain
  - Electrical and non-electrical disciplines
- Has two aspects related to physical effects
  - Across: effort like effects (voltage, velocity, temperature, etc.)
  - Through: flow like effects (current, force, heat flow rate, etc.)
- A nature defines the types of the across and through quantities incident to a terminal of the nature
- A scalar nature additionally defines the reference terminal for all terminals whose scalar subelements belong to the scalar nature
- A nature can be composite: array or record
  - All scalar subelements must have the same scalar nature
- No predefined natures in VHDL 1076.1

Electrical Systems Environment

```
package electrical_system is
    subtype voltage is REAL tolerance "default_voltage";
    subtype current is REAL tolerance "default_current";
    subtype charge  is REAL tolerance "default_charge";
    nature electrical is
        voltage          across          -- across type
        current          through         -- through type
        electrical_ref reference;        -- reference terminal
    alias ground is electrical_ref;
    nature electrical_vector is
        array(NATURAL range <>) of electrical;
end package electrical_system;
```

- Assume package is compiled into a library Disciplines

NATURES

- Physical disciplines or energy domains
  - VHDL-AMS not limited to electrical quantities

Nature	Effort	Flow
electrical	voltage	current
thermal	temperature	heat flow rate
kinematic_v	velocity	force
rotational_omega	angular velocity	torque
fluidic	pressure	volume flow rate

```
nature electrical is
 voltage across             -- across type is 'voltage'
 current through            -- through type is 'current'
 electrical_ground reference; -- reference terminal name
```

Quantities
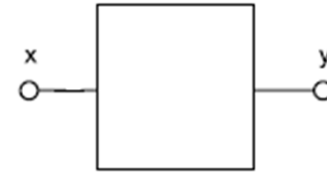
- For analog modeling

- **Continuous** time or frequency waveforms

- Quantities:
  - Free Quantities (for signal flow modeling)
  - Branch Quantities (for modeling of conservative energy systems)
  - Source Quantities (for frequency and noise modeling)

  Syntax: quantity quantity_name {,…}: subtype_indication [:= expression];

Free Quantity

- To break down complex equations into manageable pieces or for data flow description
    - Auxiliary variable

- Quantity name(s), double subtype and optionally an initial value expression

**quantity** **Vint : real := 5.0 ;**

Branch Quantity

- Branch between two terminals

- Constraints between branches that share terminal (Kirchhoff's laws in electrical domain)

- Branch quantities are used by equations of models
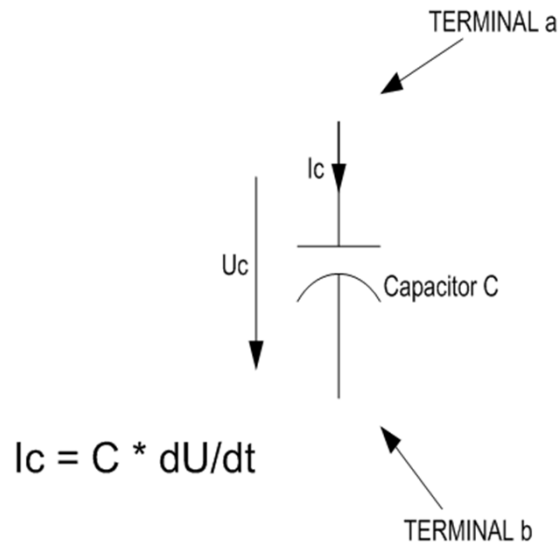
terminal anode, cathode: electrical;
quantity battery_voltage **across** battery_current **through** anode **to** cathode;
quantity leakage_voltage **across** battery_current **through** anode;
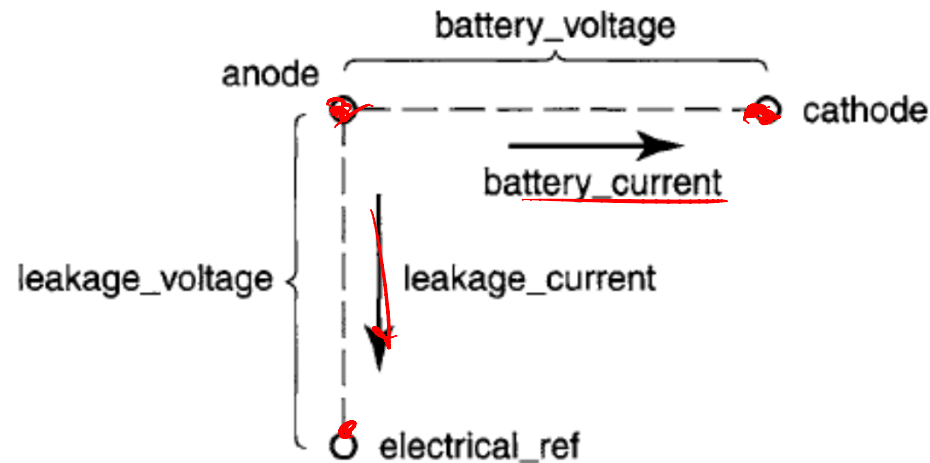
Branch Quantity

- Example of a branch quantity

TERMINAL a

Ic

Uc

Capacitor C

$Ic = C * dU/dt$

TERMINAL b

Ic is the **flow** through the terminals a and b.

- quantity Uc across Ic through a to b;

Uc is defined as the **difference** between terminals a and b

```
library ieee;
use ieee.electrical_systems.all;

-- ENTITY DEFINITON OF IDEAL CAPACITOR
entity capacitor  is
    generic ( cap:real:=10.0e-5 ); -- capacity parameter
definition
    port ( terminal a: electrical;
           terminal b: electrical);
end entity capacitor;

-- ARCHITECTURE DEFINITON OF CAPACITOR
architecture bhv of capacitor is
  quantity Uc across Ic through a to b;
begin
  Ic == cap * Uc'DOT ;
end architecture bhv;
```
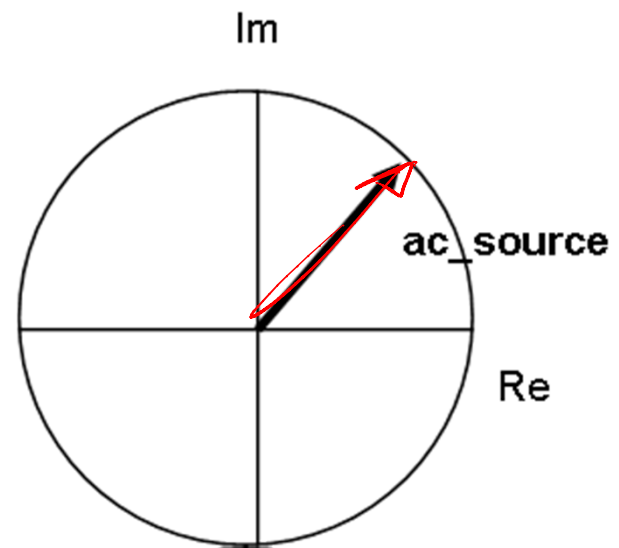
CS - ES

Source Quantity

- Specification of energy sources in frequency domain
    - E.g. voltage source

    quantity ac_source: real **spectrum** 1.0, 45.0;

Quantity Attributes

- quantity_name'**DOT**   Differential algebraic equation (DAE)
- quantity_name'**INTEG**
- quantity_name'**DELAYED[(T)]**
- quantity_name'**SLEW[(maxrise)[,(maxfall])]**   SLEW (RISE,FALL)
- quantity_name'**LTF(num,den)**   Laplace Transfer Function
- quantity_name'**ZOH(TSampl[,(init_delay])**
- quantity_name'**ZTF(num,den,t[,init_del])**   Z-Domain transfer function
- quantity_name'**ABOVE(level)**

Q'INTEG

- Integral function
- Needs to be initialized at zero time

TERMINAL a

I_L

U_L Inductor L

TERMINAL b
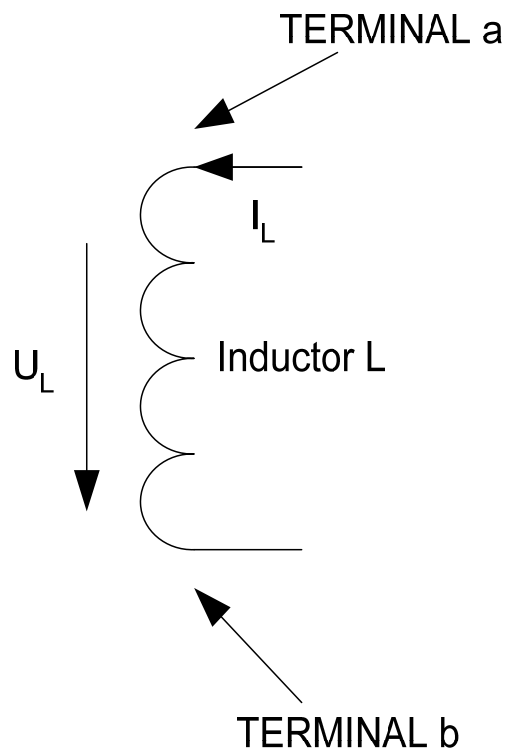
$$i_L = \frac{1}{L} \int U_L dt$$

IL == (1/L) * UL'INTEG;

Q'DELAYED

- Delay of quantity for specific time unit

Q'DELAYED([T]);

QD == Q'DELAYED(5.0e06);
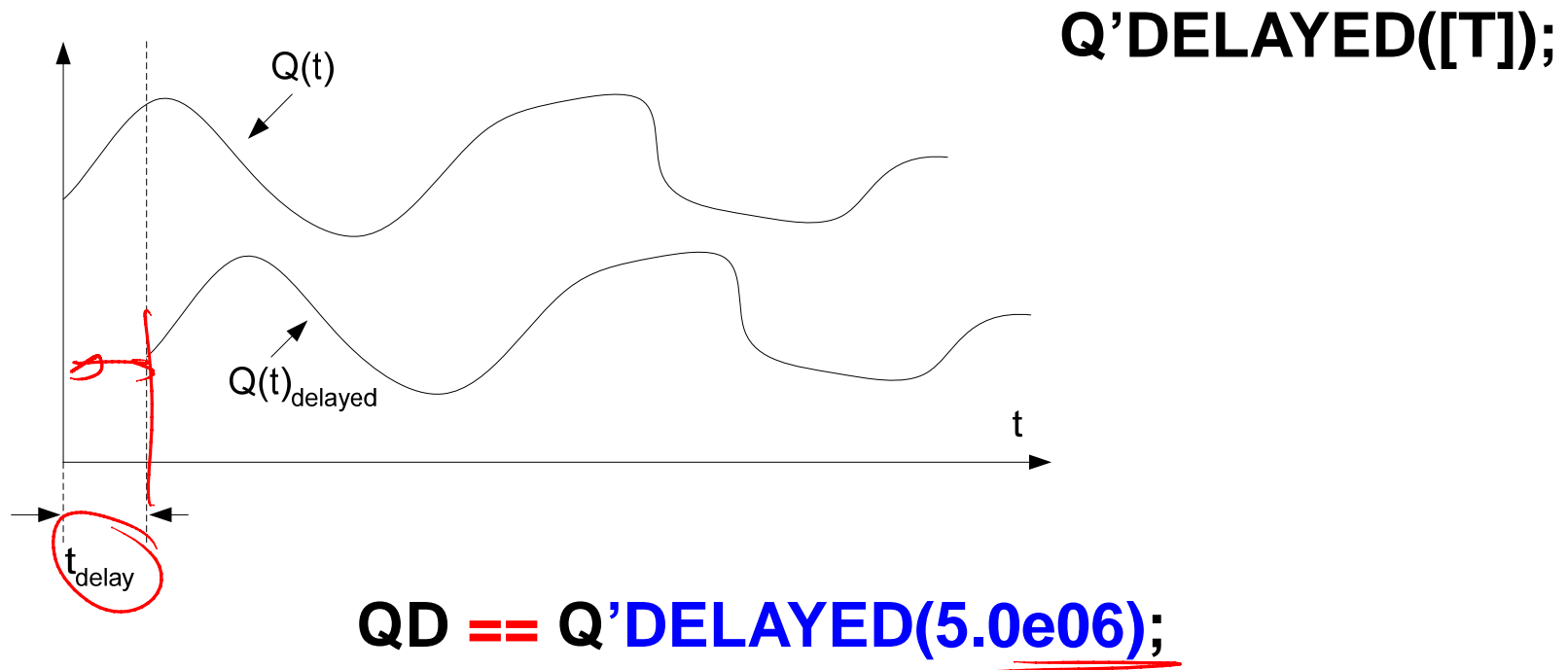
Q'LTF

- Laplace Transfer Function
- num, den – static expressions of type real_vector

$$H(s) = \frac{\sum a_k s^k}{\sum b_k s^k}$$

Q'LTF(num, den) ;

constant num : real_vector := (1.0, 2.0, 1.0);
constant den : real_vector := (1.0, 0.0, -1.0);

Q'ZOH

- ## Sample-and-Hold function
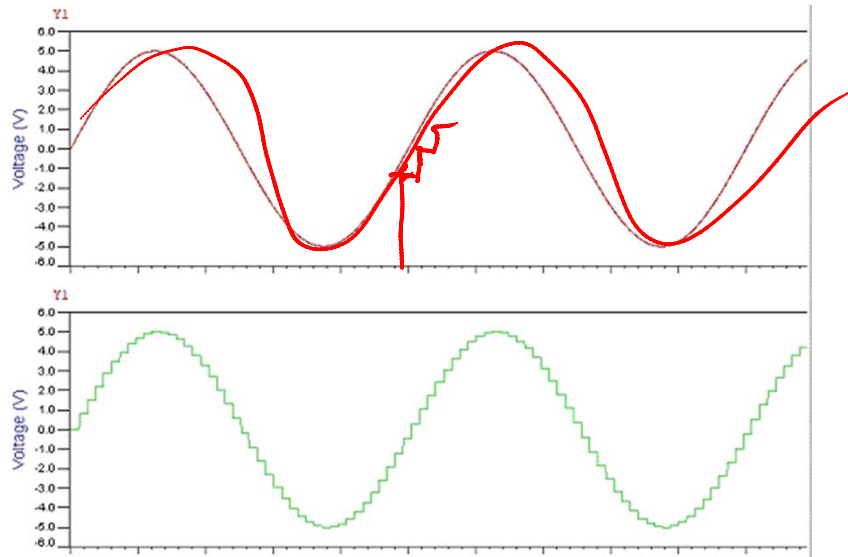
A/D D/A Converters

Q'ZOH(Tsampl [init_delay]);

- NOTE: Controlled assignment of discrete value of quantity to signal type variable –> **S <= Q'ZOH(t_sampl);**

Q'ABOVE

- Quantity – boolean signal

- Q'Above(E) = TRUE when Q > E

- Q'Above(E) = FALSE when Q < E

- No change if Q = E

- Analog to digital conversion (quantity to signal)

- **Used to trigger processes**


```
process (Q'Above(E))
begin
  if Q'Above(E) then
    S <= …
  end if;
end process;
```
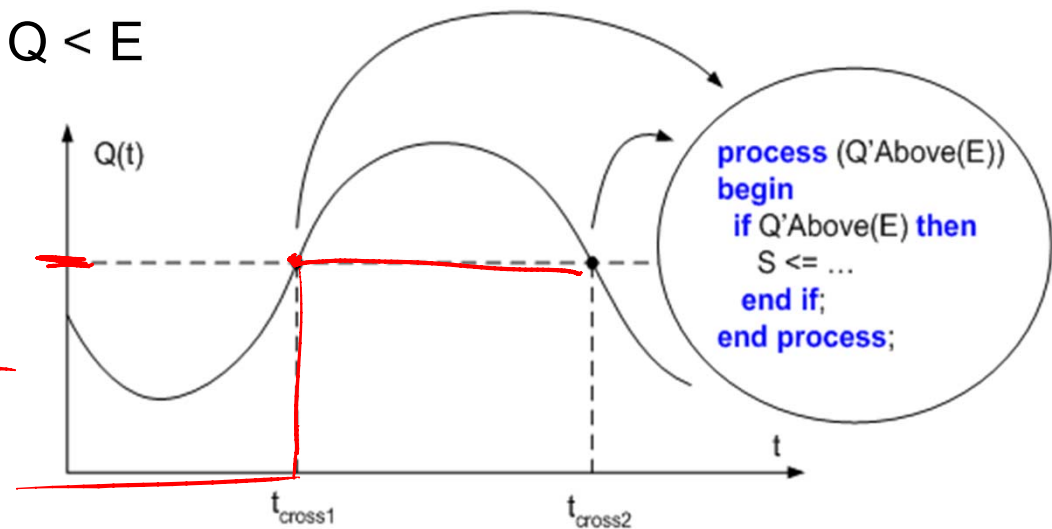
Q'Above – An example

- 1 bit ADC

```vhdl
library IEEE;
use IEEE.electrical_systems.all;

entity adc is
generic (Vref: real := 5.0 )      -- ref voltage
port (terminal inp : electrical;  -- input
      signal outp: out bit);      -- output
end entity adc;
```

```vhdl
architecture one of adc is
quantity Vin  across inp  to ref; -- input voltage
begin

 p1: process (Vin'above(Vref))
 begin
   if (Vin'above(Vref)) then
     outp <= '1';
   else
     outp <= '0';
   end if;
 end process p1;

end architecture one ;
```

Time Domain Simulation Cycle
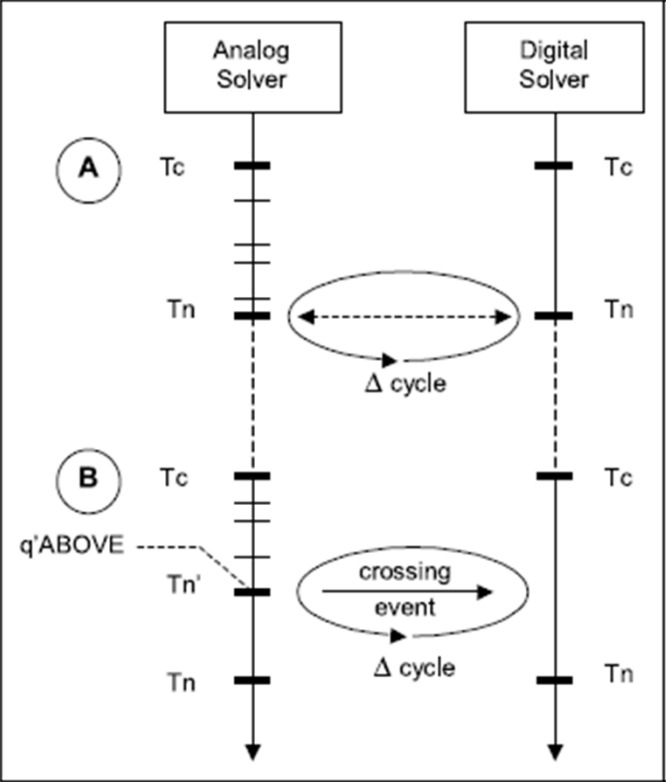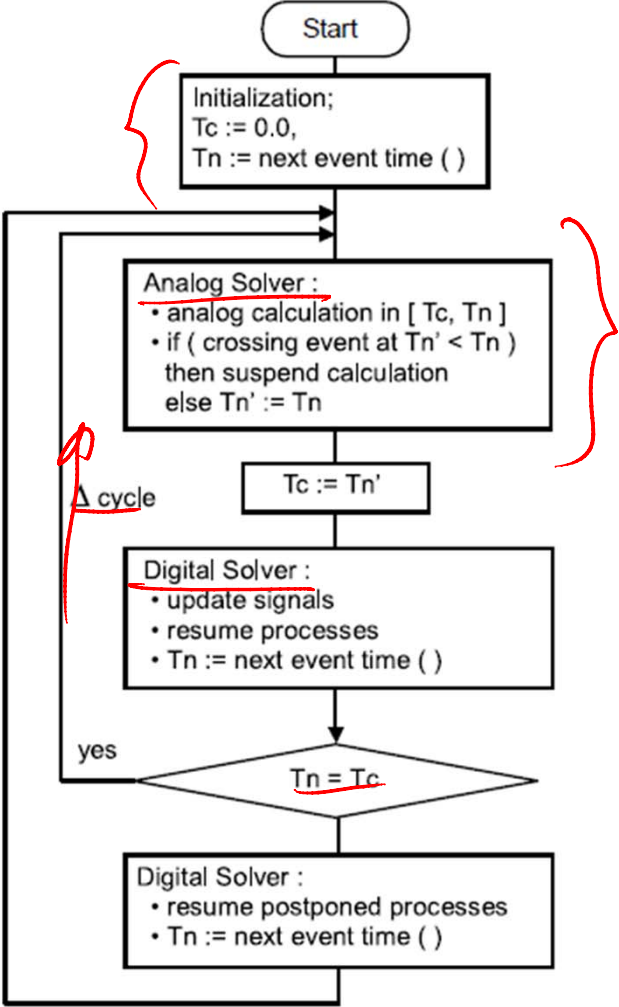
Tc – current time
Tn – next time

Example: Ideal OP amplifier

- IDEAL OP amplifier

op_inp: electrical

V_inp

+

op_inn: electrical

V_inn

-

op_outp: electrical

V_outp

OP
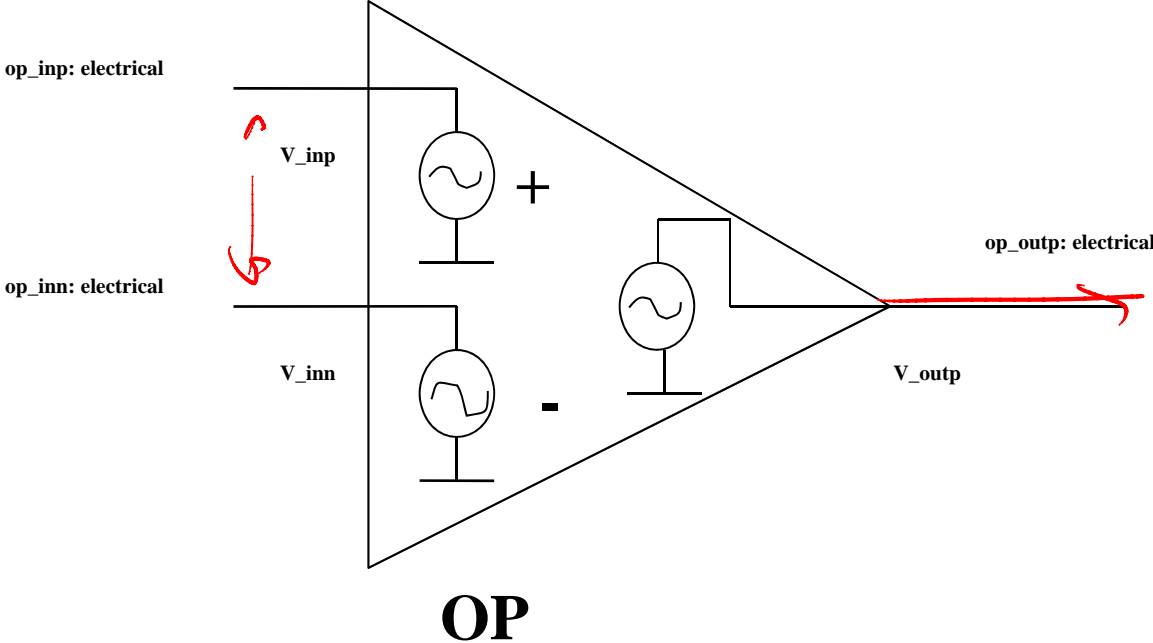
$$V\_out = Gain \bullet (V\_inp - V\_inn)$$

Example: Ideal OP amplifier

- Declaration of OP entity

```vhdl
library ieee;
use ieee.electrical_systems.all;

-- ENTITY DEFINITON OF IDEAL OP AMPLIFIER
entity op is
    generic ( gain:real:=10.0e5 ); -- gain parameter definition
    port ( terminal op_inp: electrical;
           terminal op_inn: electrical;
           terminal op_outp: electrical);
end entity op;
```

Example: Ideal OP amplifier

- Declaration of OP architecture (operational)

```
-- ARCHITECTURE DEFINITON OF IDEAL OP
architecture ideal of op is
  quantity v_inp across op_inp to electrical_ref;
  quantity v_inn across op_inn to electrical_ref;
  quantity v_outp across i_outp through op_outp to electrical_ref;
begin
  v_outp == gain * (v_inp – v_inn);    -- simultaneous statement
end architecture ideal;
```

Example: Ideal OP amplifier

- Models for resistor and voltage source
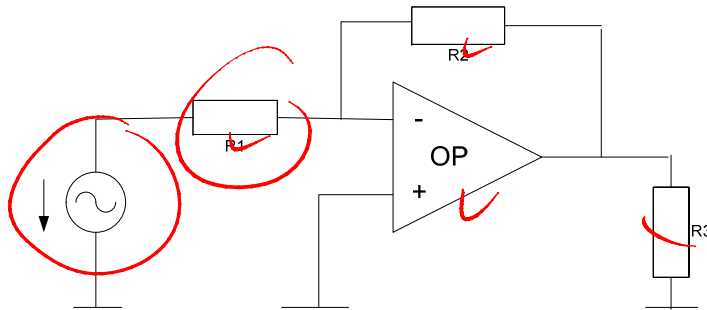

```
library IEEE;
use IEEE.electrical_systems.all;


entity resistor is
generic (res : real := 1.0e03 ); – 1kOhm
port (terminal p, n : electrical);
end entity resistor;

architecture ideal of resistor is
  quantity v across i through p to n;
begin
  v == i * res;
end architecture ideal;
```

```
library IEEE;
use IEEE.MATH_REAL.all;
use IEEE.electrical_systems.all;

entity v_sine is
    generic (
    frequency    : real:=10.0e2;        -- frequency [Hertz]
    amplitude : real:=5.0;          -- amplitude [Volts]
    phase     : real := 0.0;     -- initial phase [Degrees]
    offset    : real := 5.0);         -- DC value [Volts]
    port (  terminal pos, neg : electrical);
end entity v_sine;

architecture ideal of v_sine is
quantity v across i through pos to neg;
quantity phase_rad : real;
quantity ac_spec   : real spectrum 1.0,0.0;
begin
  phase_rad == math_2_pi *(freq * NOW + phase / 360.0);
  if domain = quiescent_domain or domain = time_domain use
   v == offset + amplitude * sin(phase_rad) * EXP(-NOW);
  else
   v == ac_spec;          -- used for Frequency (AC) analysis
  end use;
end architecture ideal;
```

Example: Ideal OP amplifier

```
use work;
-- ENTITY DEFINITON OF TEST-BENCH
entity testbench is
end entity testbench;
-- ARCHITECTURE DEFINITON OF IDEAL TEST-BENCH
architecture bhv of testbench is
    terminal sin,r12,r23:electrical;
  begin
    -- sinus source
    sinus1: entity sinus(ideal)
            generic map (amplitude=>5.0, frequency=>10.0e3, phase=>0.0);
            port map (pos=>sin, neg=>electrical_ref);
-- resistor R1
  res1: entity res(ideal)
          generic map (res=>100.0e3);
          port map (p=>sin, n=>r12);
```

```
-- OP 1
  op1: entity op(ideal)
          generic map (gain=>10.0e4);
          port map (inn=>r12,
           inp=>electrical_ref,
           outp=>r23);
-- resistor R2
  res2: entity res(ideal)
          generic map (res=>50.0e3);
          port map (p=>r12,
           n=>r23);
-- resistor R3
  res3: entity res(ideal)
          generic map (res=>10.0e3);
          port map (p=>r23,
           n=>electrical_ref);
end architecture bhv;
```

terminals
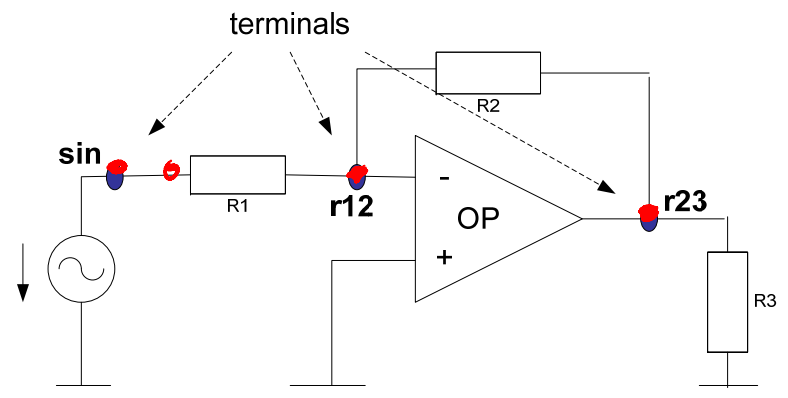
VHDL-AMS EDA Tools

- MentorGraphics
  - ADVanceMS
  - SystemVision

- Cadence
  - SimVision

- Support different subsets
  of the language

Productivity Gap

complexity [gates]

- ▲ memory
- ● processors

gates / chip

256M
64M
4M
1M
256K
64K
16K
4K
1K

486DX
386DX
8088
8080
8008

Pentium

productivity gap

gates / day

1970   1980   1990   2000   time

higher abstraction level

IP re-use

hardware software co-design

design automation

(source: MEDEA Design Automation Roadmap 1999)
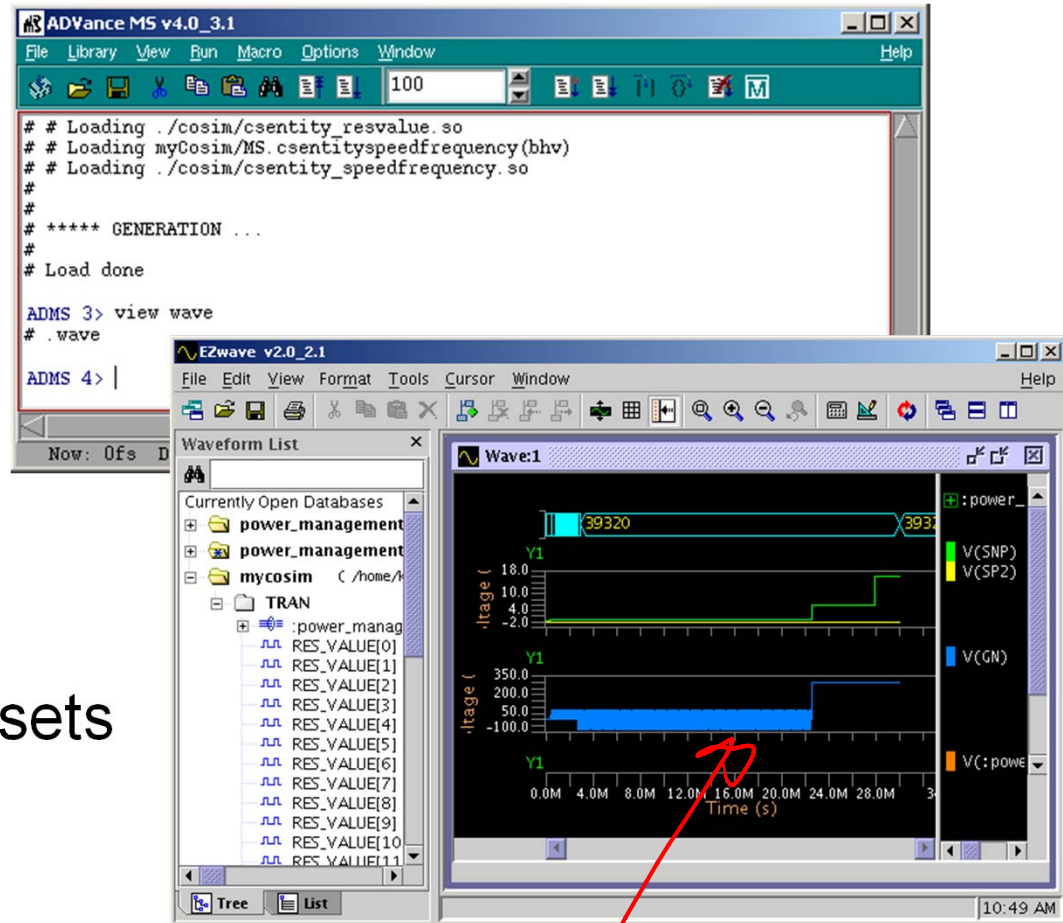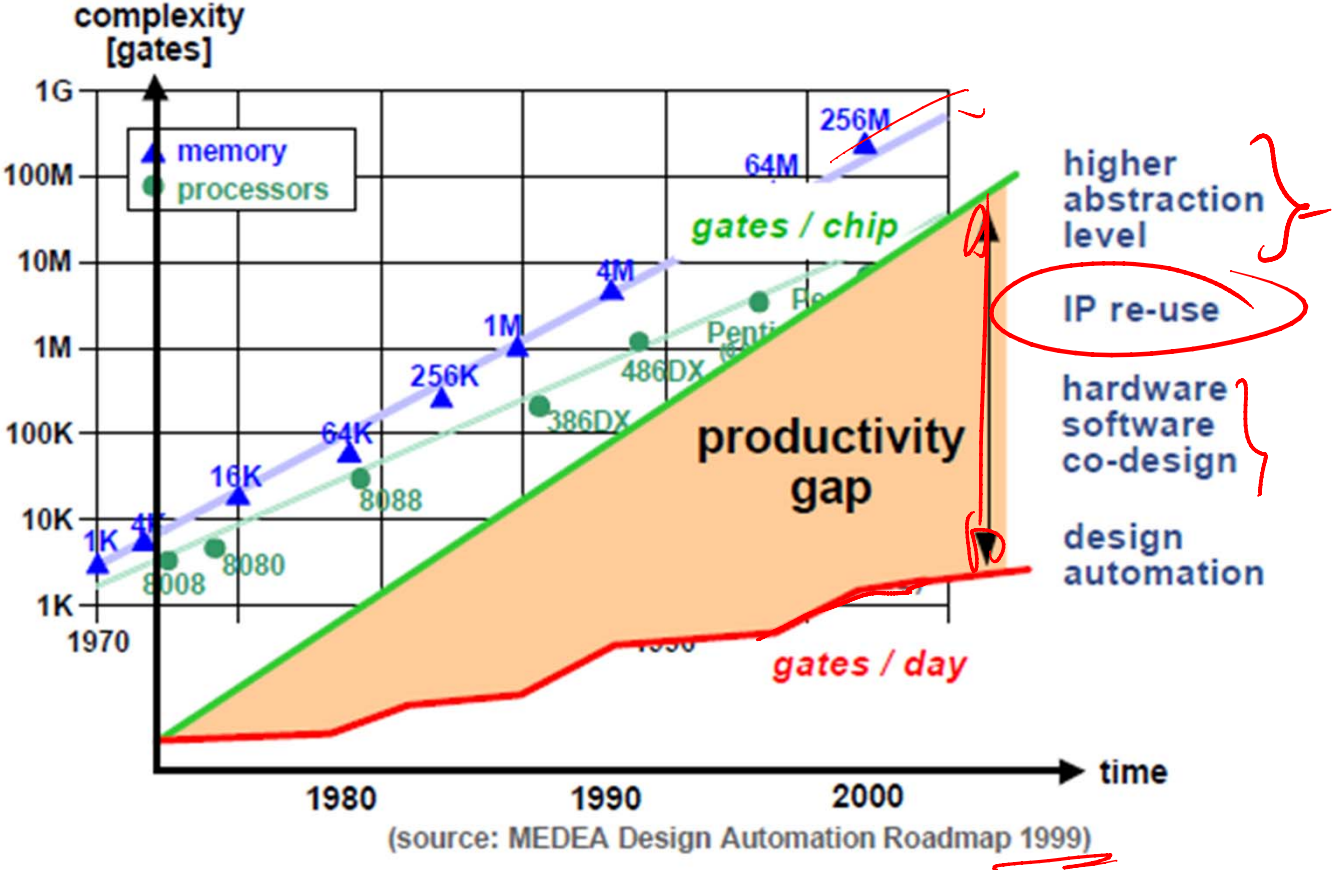
Introduction

The followings are not supported by native C/C++

- Hardware style communication
  - Signals, protocols, etc.
- Notion of time
  - Cycle/clock, delay, time sequenced op., etc.
- Concurrency
  - HW operates in parallel.
- Reactivity
  - HW responds to stimuli.
- Hardware data types
  - Bits, bit-vector types, multi-valued logic type, and so on.

SystemC is modeling platform consisting of a set of C++ class library, plus a simulation kernel that supports hardware modeling concepts at the system level, behavioral level and register transfer level.

SystemC features

- Processes $\rightarrow$ for concurrency
- Clocks $\rightarrow$ for time
- Hardware data types $\rightarrow$ bit vectors, 4-valued logic, ….
- Waiting and watching $\rightarrow$ for reactivity
- Modules, ports, and signals $\rightarrow$ for hierarchy
- Channel, interface, and event $\rightarrow$ abstract communications

- Simulation support
- Support of multiple abstraction levels and iterative refinement

CS - ES

SystemC V2.0 language structure

Upper layers are built cleanly on lower layers.

Lower layers can be used without upper layers.

Standard Channels for Various MOCs
Kahn Process Networks
Static Dataflow, etc.

Methodology-Specific Channels
Master/Slave Library
etc.

Elementary Channels
Signal, Timer, Mutex, Semaphore, Fifo, etc.

Core Language
Modules
Ports
Processes
Interfaces
Channels
Events

Data Types
Logic Type (01XZ)
Logic Vectors
Bits and Bit Vectors
Arbitrary Precision Integers
Fixed Point Numbers
C++ Built-In Types (int, char, double, etc.)
C++ User-Defined Types

C++ Language Standard

Comparison: SystemC - VHDL

		VHDL	SystemC
➤	Hierarchy	entity	module
➤	Connection	port	port
➤	Communication	signal	signal
➤	Functionality	process	process
➤	Test Bench		object orientation
➤	System Level		channel, interface, event
			abstract data types
➤	I/O	simple file I/O	C++ I/O capabilities

SystemC system

- A system consists of a set of concurrent processes that describe functionality.

- Processes communicate with each other through channels.

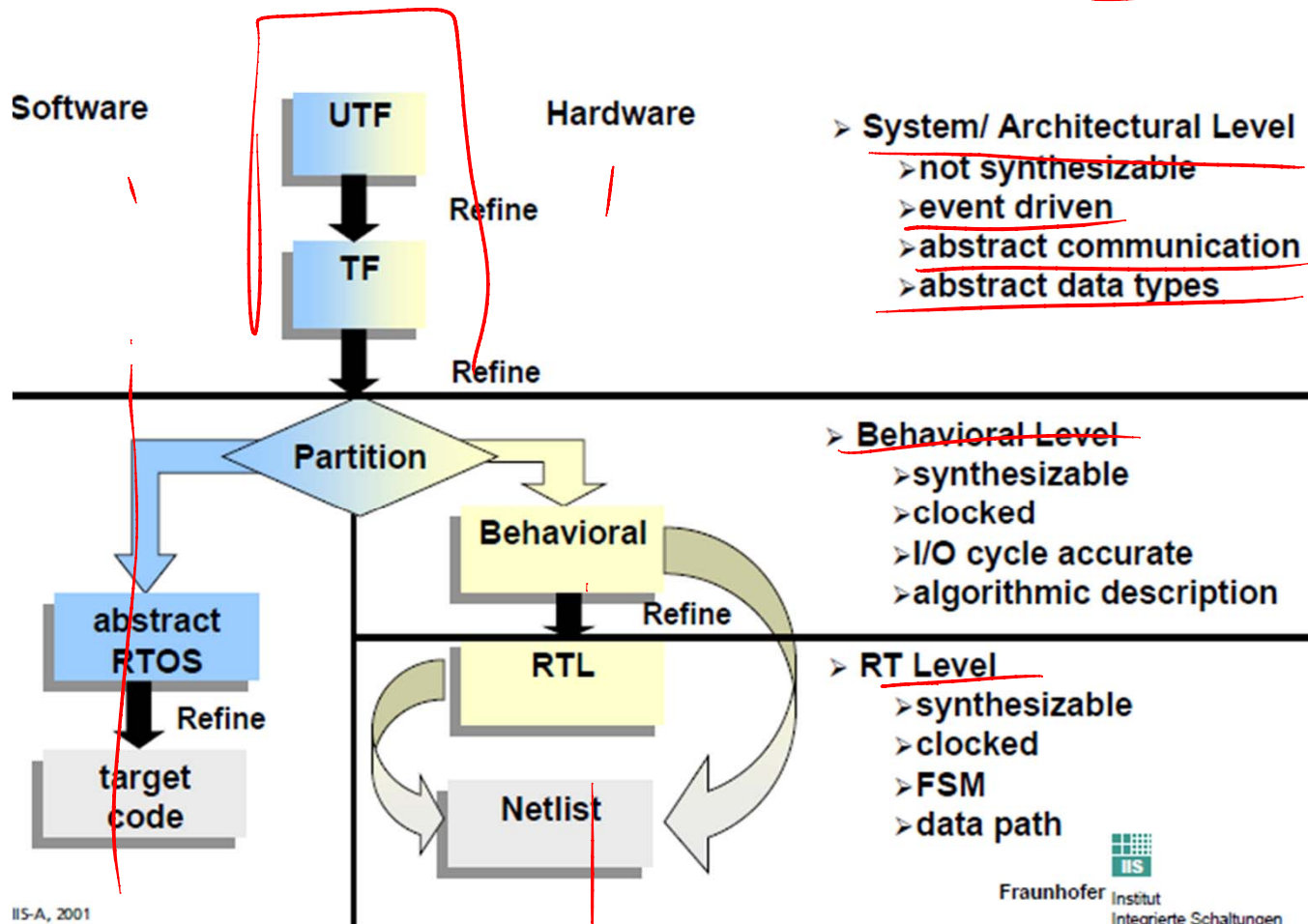- Processes can be combined into modules to create hierarchy.
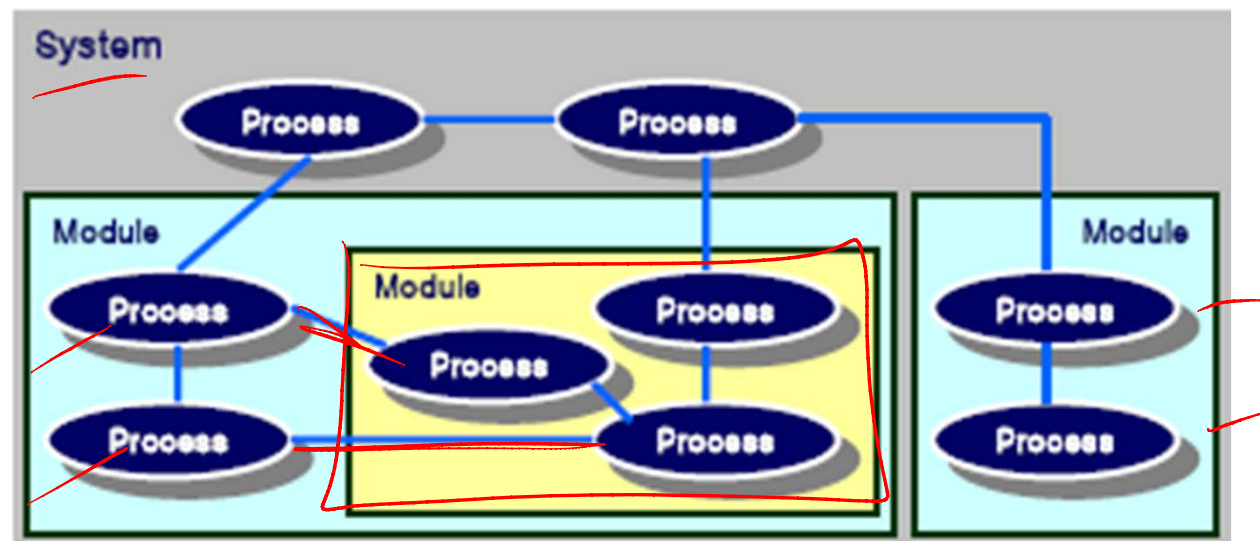
SystemC design flow

Untimed functional (UTF)
A network of HW/SW neutral modules executing in zero times and communicating with abstract channels.

Timed Functional (TF)
A network of modules executing in some defined times and communicating with abstract channels. Allows allocation of time to behavioral blocks (using wait(delay))

Software | UTF | Hardware
Refine
TF
Refine

Partition

abstract RTOS
Refine
target code

Behavioral
Refine
RTL
Netlist

> System/ Architectural Level
  > not synthesizable
  > event driven
  > abstract communication
  > abstract data types

> Behavioral Level
  > synthesizable
  > clocked
  > I/O cycle accurate
  > algorithmic description

> RT Level
  > synthesizable
  > clocked
  > FSM
  > data path

IIS-A, 2001

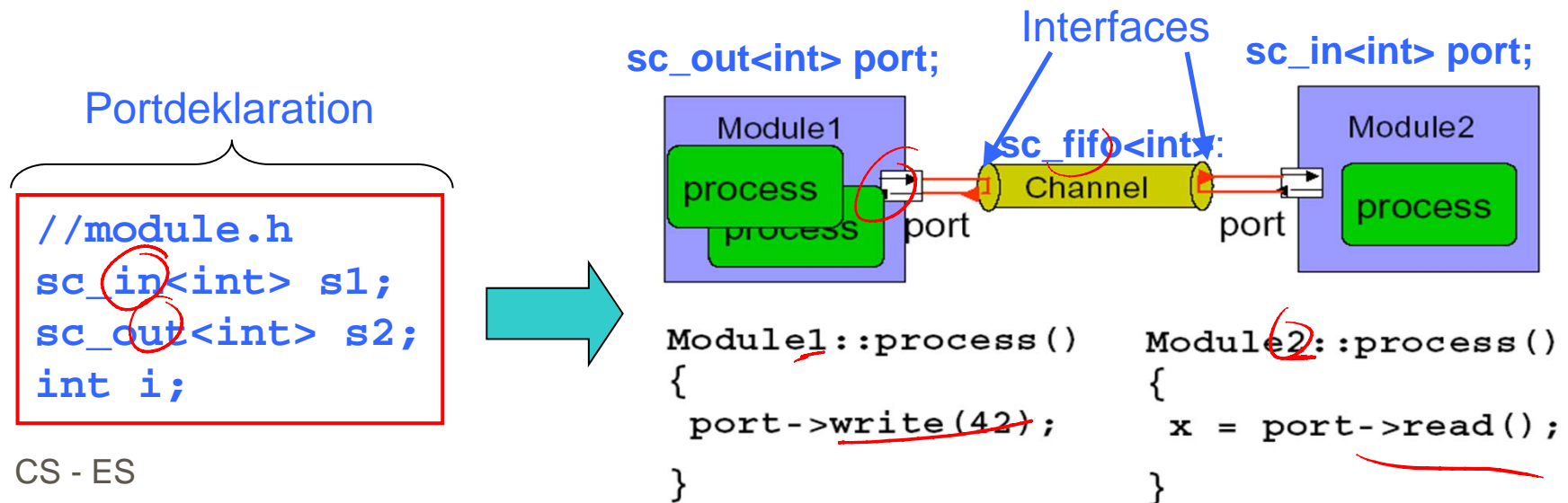Fraunhofer Institut Integrierte Schaltungen

CS - ES

- 55 -

Modules

- Modules are basic building blocks of a SystemC design
- A module contains processes (→ functionality)
- and/or sub-modules (→hierarchical structure)

```
SC_MODULE( module_name ) {
    // Declaration of module ports
    // Declaration of module signals
    // Declaration of processes
    // Declaration of sub-modules
    SC_CTOR( module_name ) { // Module constructor
    // Specification of process type and sensitivity
    // Sub-module instantiation and port mapping
    }
    // Initialization of module signals
};
```

Ports (1)

- 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

- Signals are used to connect module ports allowing modules to communicate

- Very similar to ports and signals in VHDL

Portdeklaration

```
//module.h
sc_in<int> s1;
sc_out<int> s2;
int i;
```

sc_out<int> port;

Interfaces

sc_in<int> port;

Module1

process

process

sc_fifo<int>:

Channel

port

port

Module2

process

```
Module1::process()
{
  port->write(42);
}
```

```
Module2::process()
{
  x = port->read();
}
```

CS - ES

Ports and Signals

- How to read and write a port ?

  - Methods read( ); and write( );

- Examples:

  - in_tmp = in.read( ); //reads the port in to in_tmp

  - out.write(out_temp); //writes out_temp in the out port

Signals

- sc_signal<T>
    - Signal are also used for connecting two modules ports in parent module.

- sc_fifo<T>
    - sc_fifo is a predefined primitive channel intended to model the behavior of a fifo, that is, a first-in first-out buffer.

- sc_buffer<T>
    - like sc_fifo (buffer size 1)
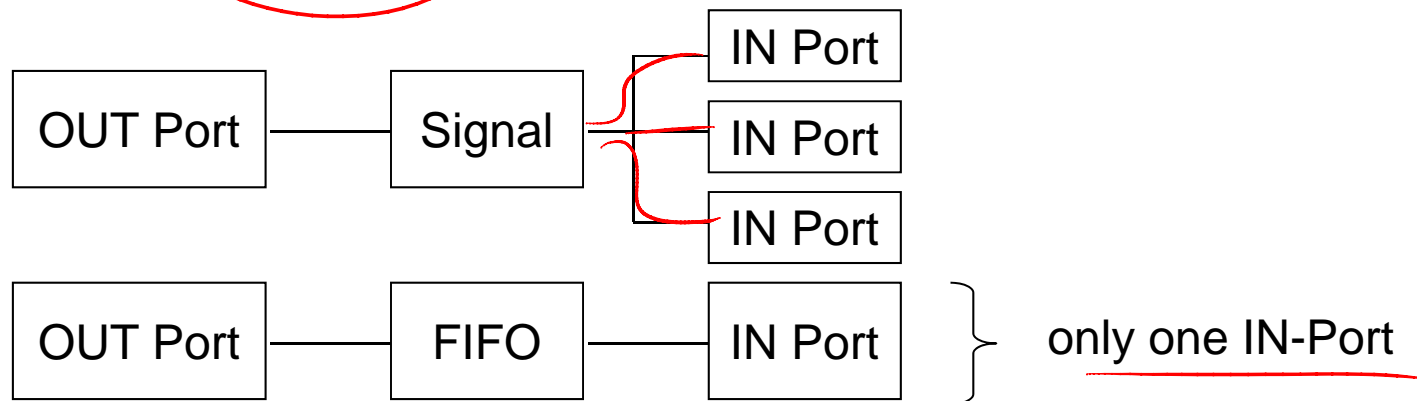
Events

- data type: **sc_event**

- functionality:
  - declaration: **sc_event my_event;**
  - **my_event.notify();** // immediately
  - **my_event.notify(SC_ZERO_TIME);** // next delta-Zyklus
  - **my_event.notify(10, SC_NS);** // after 10 Ns

- waiting on event: **wait(my_event);**
  - process suspended until new event

Processes

- **Process Semantics**
  - Encapsulates functionality
  - Basic unit of concurrent execution
  - Not hierarchical
- **Process Activation**
  - Processes have sensitivity lists
  - Processes are triggered by events on sensitive signals
- **Process Types**
  - Method (SC_METHOD)
    - asynchronous block, like a sequential function
  - Thread (SC_THREAD)
    - asynchronous process
  - Clocked Thread (SC_CTHREAD)
    - synchronous process

Processes

	SC_METHOD	SC_THREAD	SC_CTHREAD
triggered	by signal events	by signal events	by clock edge
infinite loop	no	yes	yes
execution suspend	no	yes	yes
suspend & resume	-	wait()	wait() wait_until()

construct & sentisize method	SC_METHOD(p);
	
	sensitive(s);
	sensitive_pos(s);
	sensitive_neg(s);
		SC_THREAD(p);
		
		sensitive(s);
		sensitive_pos(s);
		sensitive_neg(s);
			SC_CTHREAD
			(p,clock.pos());
			
			SC_CTHREAD
			(p,clock.neg());

modeling example (hardware)	combinational logic	sequential logic at RT level (asynchronous reset, etc.)	sequential logic at higher design levels

SC_METHOD

```
SC_MODULE( plus ) {
   sc_in<int>  i1;
   sc_in<int>  i2;
   sc_out<int> o1;

   void do_plus();

   SC_CTOR( plus ) {
      SC_METHOD( do_plus );
      sensitive << i1 << i2;
   }
};
```

```
void plus::do_plus() {
   int arg1;
   int arg2;
   int sum;

   arg1 = i1.read();
   arg2 = i2.read();
   sum = arg1 + arg2;
   o1.write(sum);
}
```

```
void plus::do_plus() {
   o1 = i1 + i2;
}
```

SC_THREAD

```cpp
SC_MODULE( plus ) {
    sc_in<int>  i1;
    sc_in<int>  i2;
    sc_out<int> o1;

    void do_plus();

    SC_CTOR( plus ) {
        SC_THREAD( do_plus );
        sensitive << i1 << i2;
    }
};
```

```cpp
void plus::do_plus() {
    int arg1;
    int arg2;
    int sum;

    while ( true )  {
        arg1 = i1.read();
        arg2 = i2.read();
        sum = arg1 + arg2;
        o1.write(sum);

        wait();
    }
}
```

SC_CTHREAD

```cpp
SC_MODULE( plus ) {
  sc_in_clk   clk;

  sc_in<int>  i1;
  sc_in<int>  i2;
  sc_out<int> o1;

  void do_plus();

  SC_CTOR( plus )  {
    SC_CTHREAD( do_plus, clk.pos() );
  }
};
```

```cpp
void do_plus() {
   int arg1;
   int arg2;
   int sum;

   while ( true )  {
      arg1 = i1.read();
      arg2 = i2.read();
      sum = arg1 + arg2;
      o1.write(sum);

      wait();
   }
}
```

CS - ES

Static Sensitivity

- Explicit sensitivity list in constructor
- Process declared as SC_METHOD()
- Connectivity is known only after construction, EventFinder find the relevant event and connect to it

```
sensitive << clk.pos();
```

Sensitivity with Clock

Ctrl.h

```
SC_MODULE (Ctrl) {
  AMBA::AHBInitiator_outmaster_port<32,32> P1;
  sc_in_clk                                clk;

  SC_CTOR(Ctrl)
  : P1("P1")
  {
    SC_METHOD(compute);
    sensitive << clk.pos();
    dont_initialize();
  }

  void compute(void);
};
```

Process is triggered every cycle, need to test bus availability

```
void Ctrl::compute() {
  if (P1.getTransaction()) {
    P1.Transaction->setAddress(myAddr);
    P1.Transaction->setAccessSize(32);
    P1.Transaction->setType(tlmWriteAtAddress);
    P1.Transaction->setWriteData(myData);
    P1.sendTransaction();
  }
}
```

Dynamic Sensitivity

- Process declared as SC_THREAD()
- Use wait() statement in the process

```
wait( P1.getSendAddrTrfEvent() );
```

Example: Dynamic Sensitivity

Ctrl.h

```
SC_MODULE (Ctrl) {
  AMBA::AHBInitiator_outmaster_port<32,32> P1;

  SC_CTOR(Ctrl)
  : P1("P1")
  {
    SC_THREAD(compute);
  }

  void compute(void);
};
```

Ctrl.cpp

Infinite loop with "waits"

```
void Ctrl::compute() {
  while(1) {
    wait(P1.getReceiveAddrTrfEvent());
    P1.getAddrTrf();
    ADDR = P1.AddrTrf->getAddress();
    . . .
  }
}
```
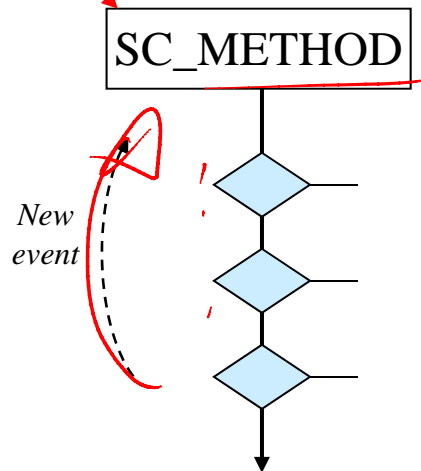
Comparison

Static

Event:
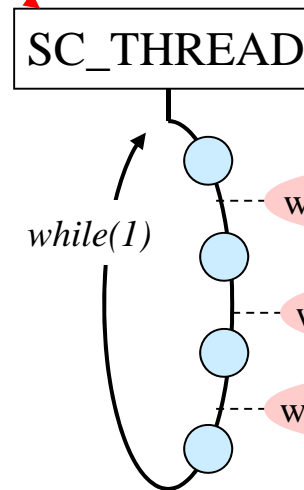- clock
- transfer

SC_METHOD

New event

Sensitivity established in the constructor:

```
SC_CTOR(myModule) {
    SC_METHOD(myProcess);
    sensitive << P1.____TrfEventFinder();
    dont_initialize();
}
```

Dynamic

Initialization
(or event: clock, transfer)

SC_THREAD

Event sequence must be correct or code will stall!

while(1)

wait(**event**)

wait(**event**)

wait(**event**)

Sensitivity embedded in the process code:

```
void myProcess(void) {
    while(1) {
        do_something();
        wait(P1.____TrfEvent());
        . . . .
```

Dynamic Sensitivity

- **wait()**
- **only for SC_THREADS and SC_CTHREADS !!**
- **wait(e1);**

- **// waiting for e1 or e2**
- **wait(e1 | e2);**

- **// waiting foe e1 and e2**
- **wait(e1 & e2);**

- **// wait 200 ns**
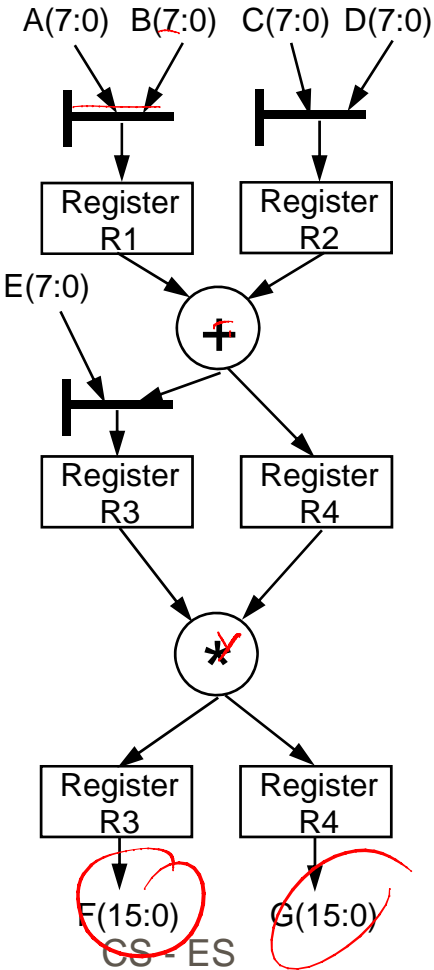- **wait(200, SC_NS);**

Behavioral VS RTL-level Modeling

- **Behavioral-level**
  - **Don't care about states, registers etc.**
  - **Use I/O-cycles**
  - **Think your design as program flow**

- **RTL-level**
  - **Map different states, registers etc.**
  - **Use clock-cycles**
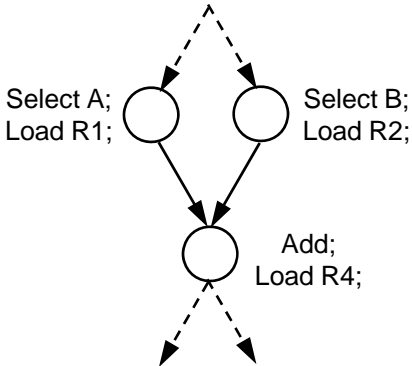 - **Think your design as finite- state-machine**

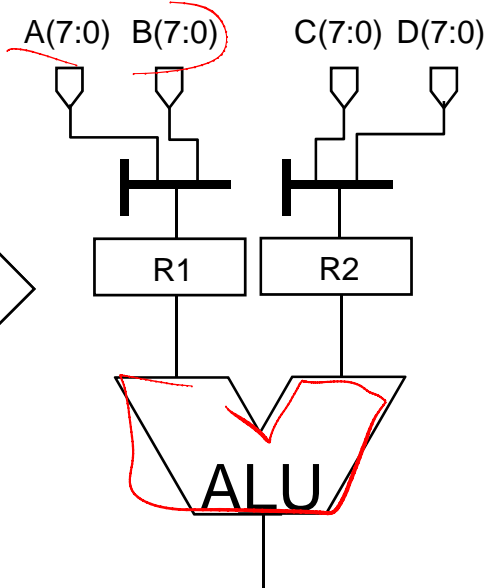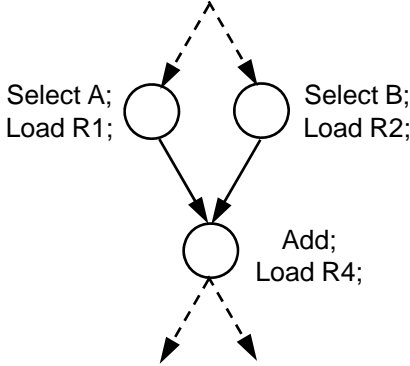# RTL Level Synthesis

Data Path Behavior

A(7:0) B(7:0)   C(7:0) D(7:0)

Register R1   Register R2

E(7:0)

+

Register R3   Register R4

*

Register R3   Register R4

F(15:0)   G(15:0)

CS - ES

Control Flow
(not all shown)

Select A;
Load R1;   Select B;
                Load R2;

Add;
Load R4;

RTL Structure
(not all shown)

A(7:0) B(7:0)   C(7:0) D(7:0)

R1   R2

ALU

RTL Control Flow
(not all shown)

Select A;
Load R1;   Select B;
                Load R2;

Add;
Load R4;

Logic Synthesis

RTL Structure
(not all shown)

A(7:0) B(7:0)   C(7:0) D(7:0)

R1   R2

ALU

RTL Control Flow
(not all shown)

Select A;   Select B;
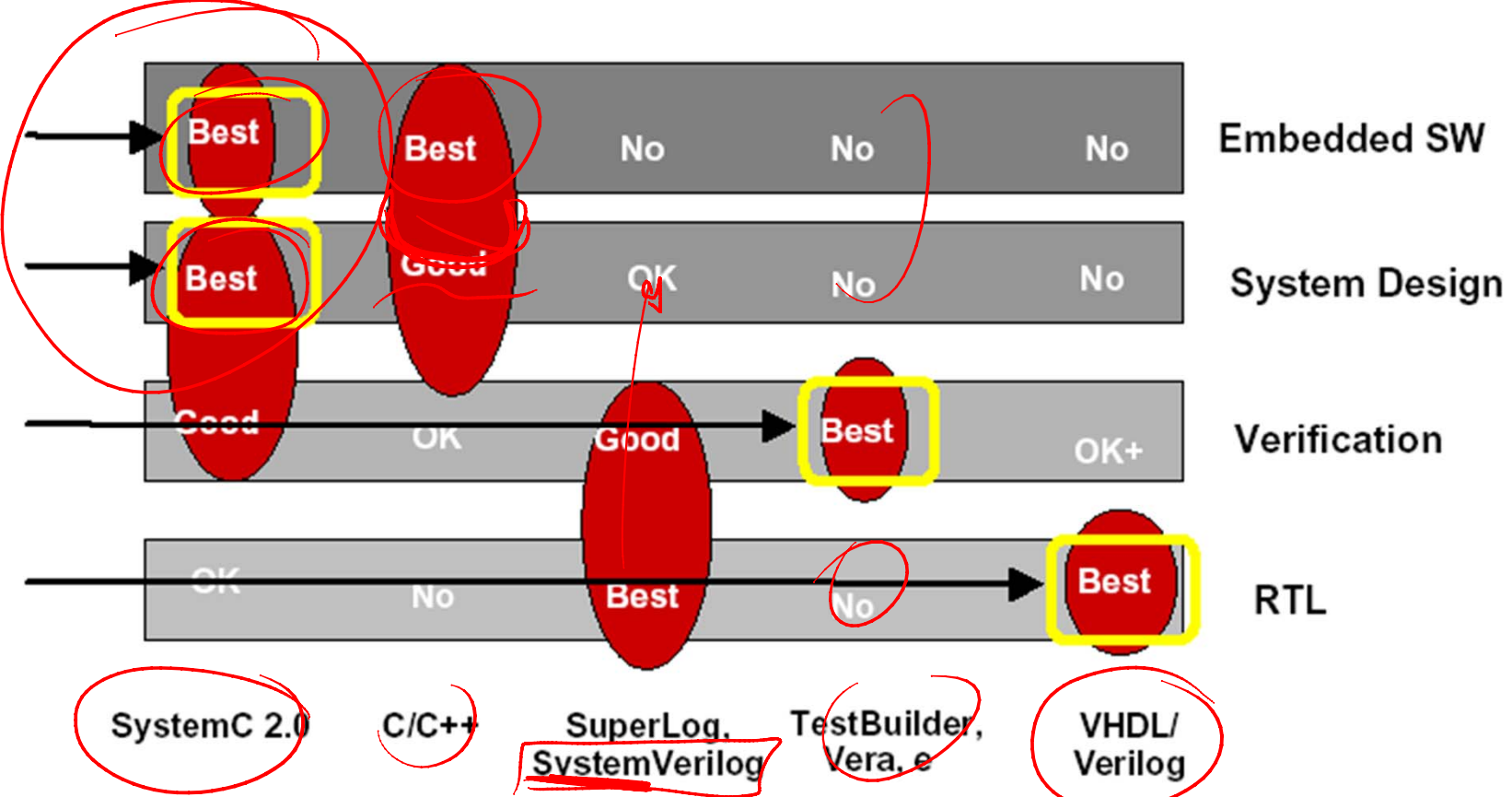Load R1;   Load R2;

Add;
Load R4;

Untimed Functional Modeling

- Model of computation is KPN
  - Communication is handled through limited size FIFO-channels with blocking read() and write() operations

- Algorithmic delays are represented by initial values stored in FIFOs
- Use modules that contain SC_THREAD processes

- No time will be present, everything advances in delta cycles
- Synchronization is implicit
  - Write access block until space is available
  - Read access block until data is available
- Caution
  - Provide initial values of FIFOs prior simulation
  - Make sure that data is produced before it is consumed

Timed functional models

- Notion of time is needed when functional models are used on lower level of abstraction

- Processing delays are done with wait(sc_time)

- Timed and untimed models can peacefully coexist and interact

- It is even possible to mix FIFO-and signal-based communication

Hardware description languages

	Embedded SW
Best, Best, No, No, No	
Best, Good, OK, No, No	System Design
Good, OK, Good, Best, OK+	Verification
OK, No, Best, No, Best	RTL

SystemC 2.0   C/C++   SuperLog, SystemVerilog   TestBuilder, Vera, e   VHDL/ Verilog

└ Sysh C -AN

→ VHDL DXS

SystemC – Transaction level modeling

Concept of TLM

- Event-driven simulation style
    - That's the name "Transaction"
    - Simulation triggered by data communication
    - Bus events
- Goal
    - High speed simulation
    - Simplify modeling
    - Early system analysis

- In a transaction-level model (TLM), the details of communication among computation components are separated from the details of computation components.

Transaction Level Modeling

- Transaction-level modeling allows faster simulations than pin-based interfaces
    - e.g. large burst-mode transfer may take many actual clock cycles, here we can use burst_read and burst_write methods
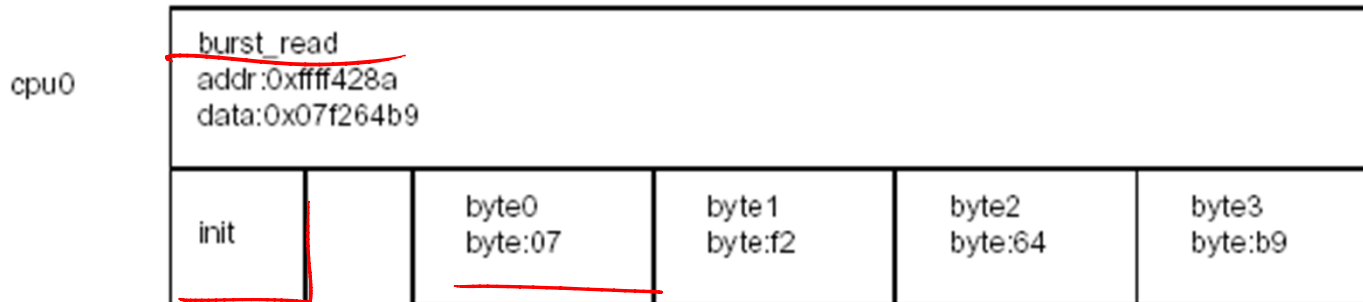
- Use transaction-level modeling when it is beneficial
    - functional modeling (untimed and timed)
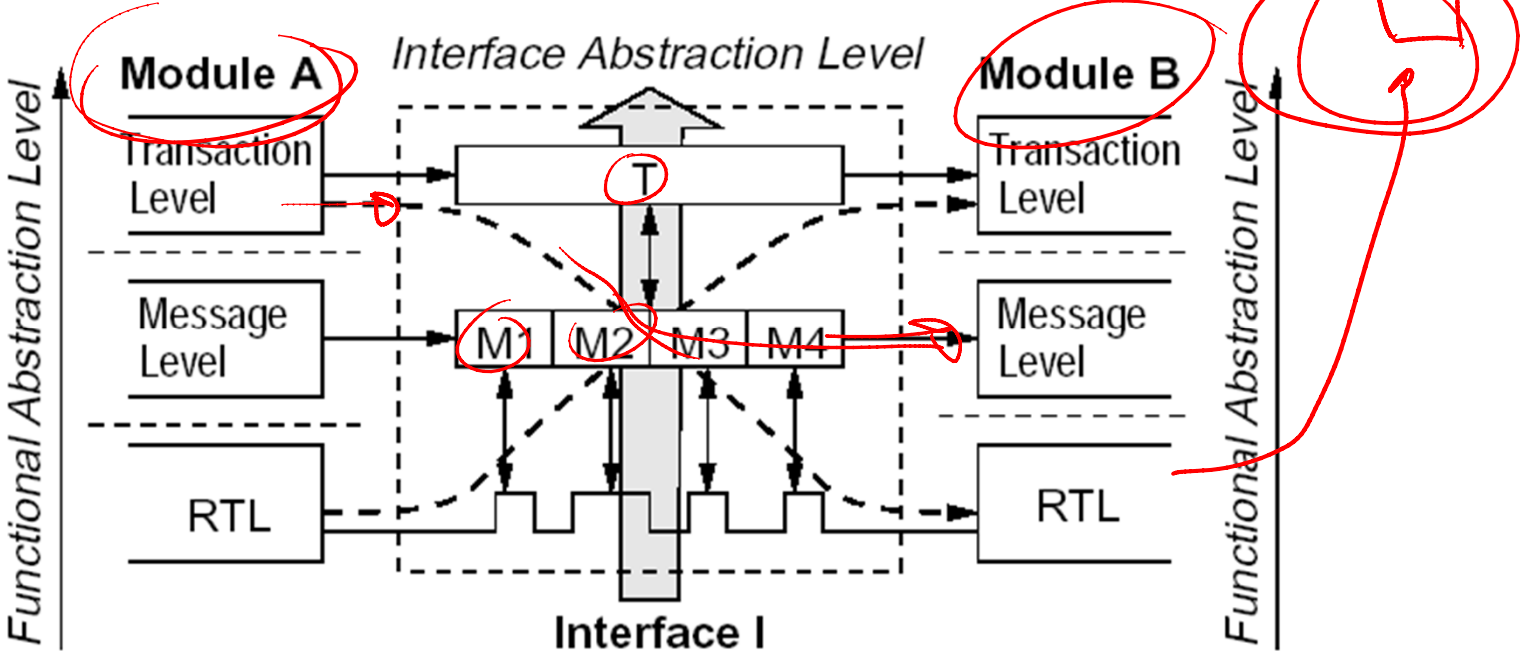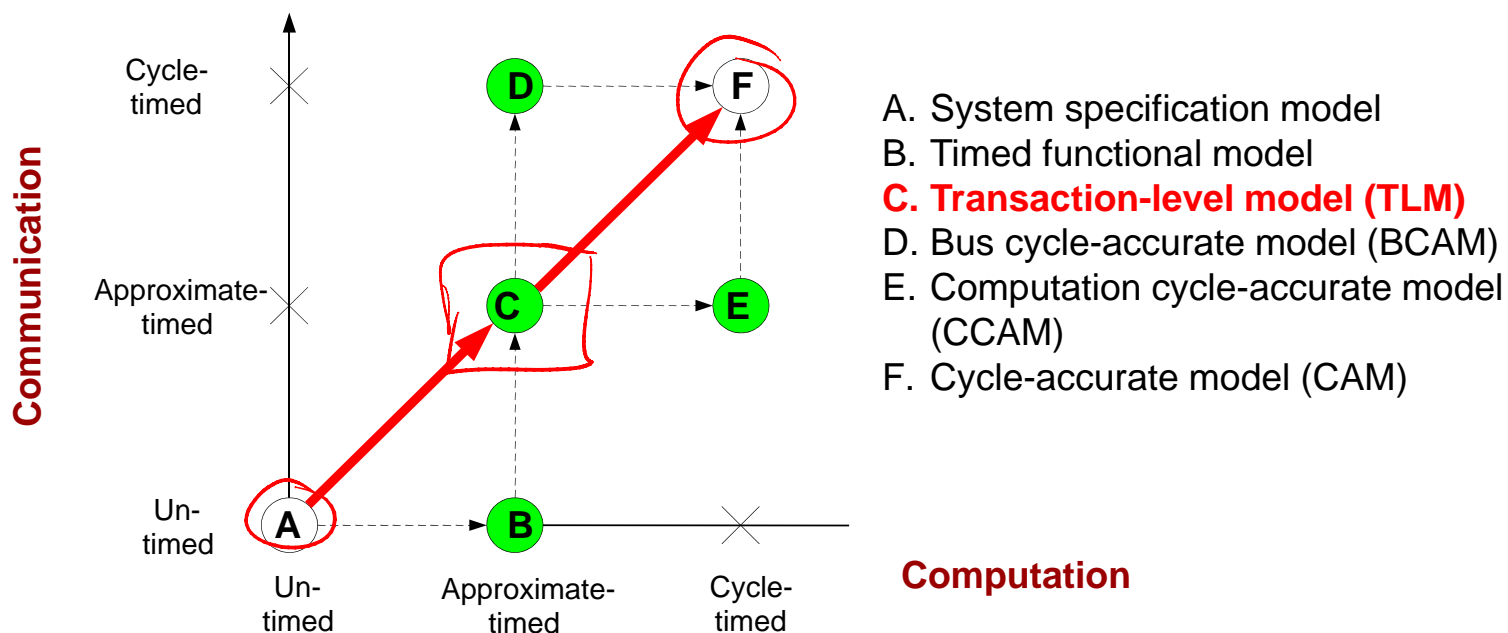   - platform modeling
    - test benches

TLM- refinement
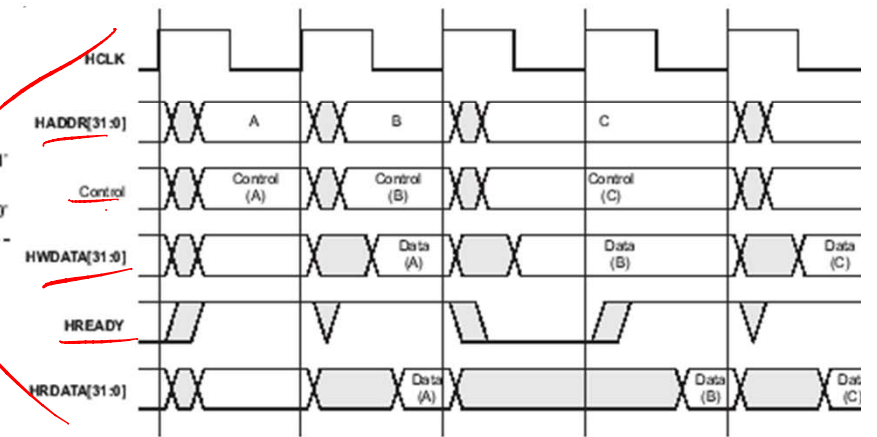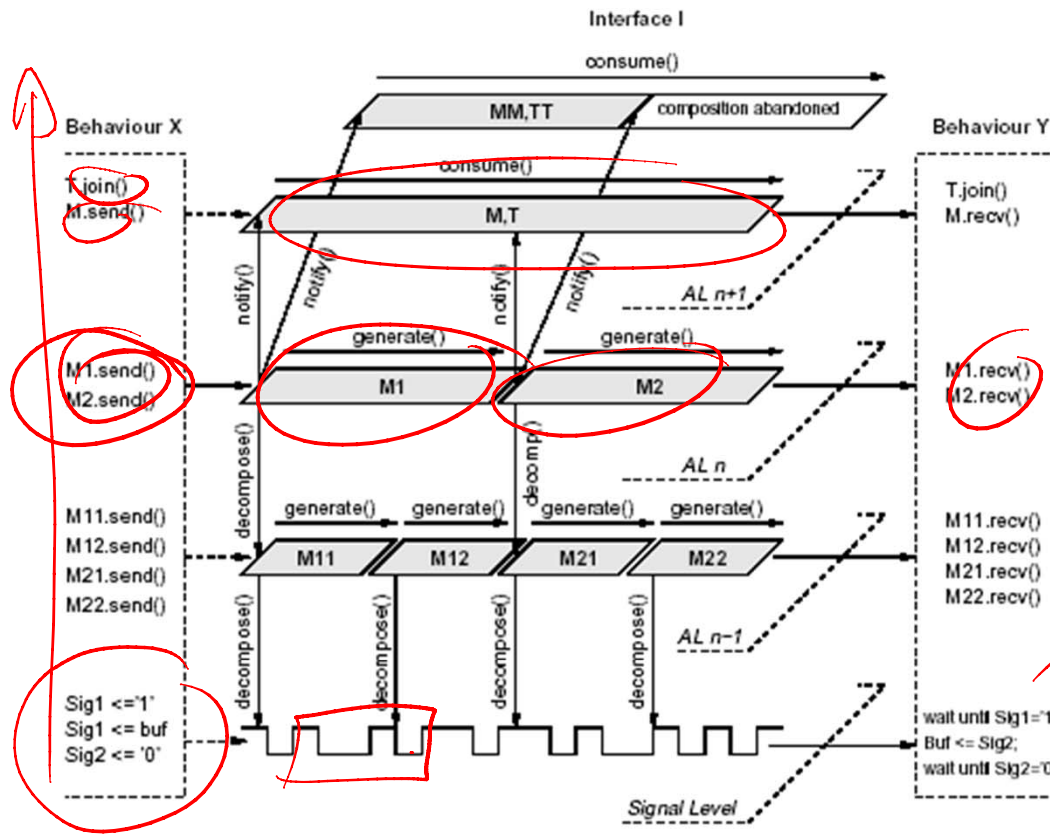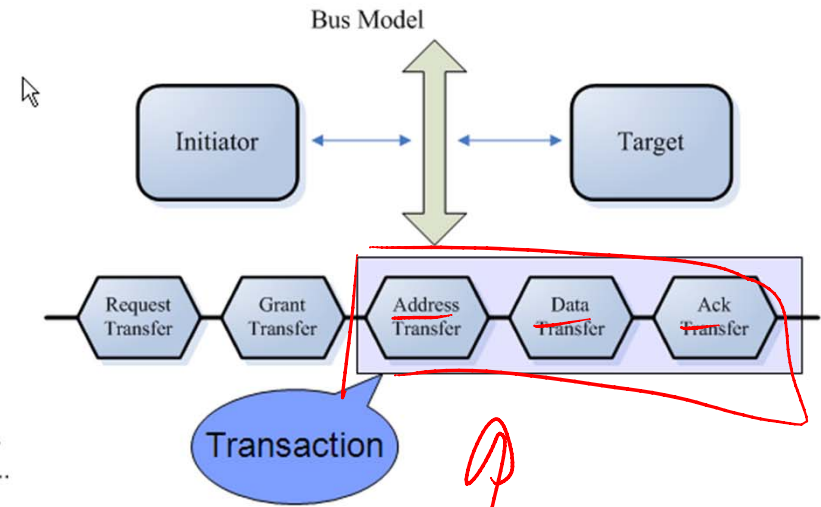
TLM- different abstraction layers

Abstraction Levels

- Abstraction based on level of detail & granularity
    - Computation and communication
- System design flow
    - Path from model A to model F

A. System specification model
B. Timed functional model
C. Transaction-level model (TLM)
D. Bus cycle-accurate model (BCAM)
E. Computation cycle-accurate model (CCAM)
F. Cycle-accurate model (CAM)

- **Design methodology and modeling flow**
    - Set of models and transformations between models

Referenzen

- **SystemC Quickreference**
  - http://comelec.enst.fr/hdl/sc_docs/systemc_quickreference.pdf

- **ASIC World SystemC Tutorial:**
  - http://www.asic-world.com/systemc/tutorial.html
  - Codebeispiele

- **SystemC Einführung**
  - SystemC Introduction, W.Yang, Dynalith Systems, 2006

- **SystemC, TLM (Transaktionsebene) Tutorials, SystemC Verifcation Library**
  - http://www.doulos.com/knowhow/systemc/

Overview of embedded systems design