VHDL
Basic Issues and Simulation Semantics

1. VHDL: History and Main Features

2. Basic Constructs

3. An Example: Behavioral and Structural Models

4. Concurrent Statements

5. Signals and the Wait Statement

6. The VHDL Simulation Mechanism

7. The Delay Mechanism

8. Resolved Signals

9. VHDL for System Synthesis

VHDL History

- The name: VHSIC Hardware Description Language

- Important dates:

  - 1983: development started with support from US government.

  - 1987: adopted by IEEE as a standard (IEEE Std. 1076 - 1987).

  - 1993: VHDL'92 adopted as a standard after revision of the initial version (IEEE Std. 1076 - 1993).

- Work is going on for the release of new revisions (e.g. including facilities for analog modeling and simulation).

Main Features

- Supports the whole design process from high to low abstraction levels:
  - system and algorithmic level
  - Register Transfer (RT) level
  - logic level
  - circuit level (to some extent)
- Suitable for specification in
  - behavioral domain
  - structural domain

Main Features (cont'd)

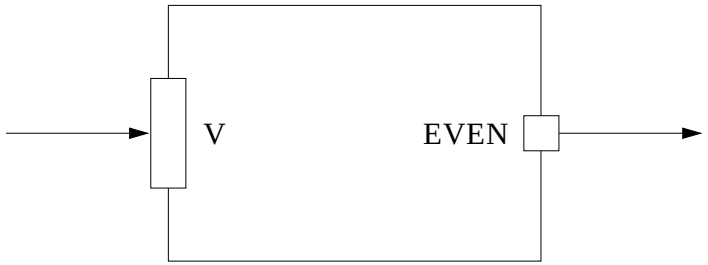
- Precise simulation semantics is associated with the language definition:
  - specifications in VHDL can be simulated;
  - the simulation output is uniquely defined and independent of the tool (VHDL implementation) and of the computer on which the tool runs.
- VHDL specifications are accepted by hardware synthesis tools.
  - Both the input and the output of the synthesis process are very often codified in VHDL.

Basic Constructs

- The basic building block of a VHDL model is the *entity.*
- A digital system in VHDL is modeled as an entity which itself can be composed of other entities.
- An entity is described as a set of *design units*:
  - **entity declaration**
  - **architecture body**
  - package declaration
  - package body
  - configuration declaration
- A design unit can be compiled separately.

An Example

A four bit parity generator




```
entity PARITY is
    port(V:in BIT_VECTOR(3 downto 0);
         EVEN:out BIT);
end PARITY;
```

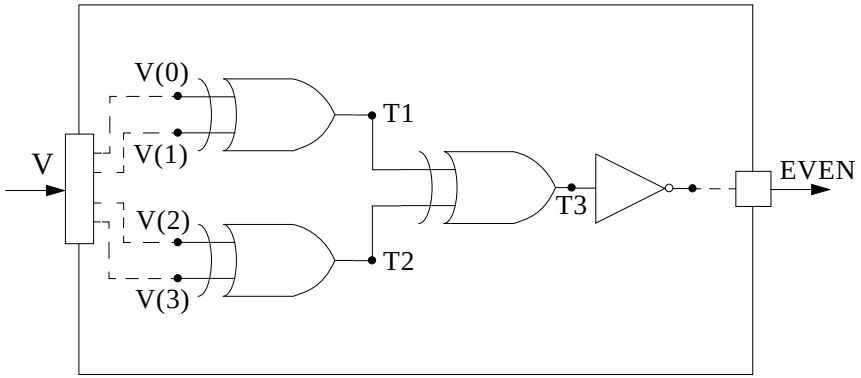
An Example (cont'd)

Architecture body for parity generator - behavioral

```
architecture PARITY_BEHAVIORAL of PARITY is
begin
  process
    variable NR_1: NATURAL;
  begin
    NR_1:=0;
    for I in 3 downto 0 loop
      if V(I)='1' then
        NR_1:=NR_1+1;
      end if;
    end loop;
    if NR_1 mod 2 = 0 then
      EVEN<='1' after 2.5 ns;
    else
      EVEN<='0' after 2.5 ns;
    end if;
    wait on V;
  end process;
end PARITY_BEHAVIORAL;
```


An Example (cont'd)

Parity generator - structural



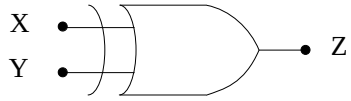
The same external interface as before; only the internal description differs.

The same entity declared on slide 7.

But another architecture body has to be attached to it.

An Example (cont'd)

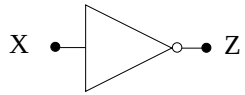
Let's first specify the building blocks




```
entity XOR_GATE is
  port(X, Y:in BIT;
       Z:out BIT);
end XOR_GATE;
```

```
architecture ARCH_XOR of XOR_GATE is
begin
  Z<=X xor Y after 1 ns;
end ARCH_XOR;
```


An Example (cont'd)




```
entity INV is
  generic(DEL: TIME);
  port(X:in BIT;
       Z:out BIT);
end INV;
```

```
architecture ARCH_INV of INV is
begin
  Z<=not X after DEL;
end ARCH_INV;
```

An Example (cont'd)

Architecture body for parity generator - structural

```
use WORK.all;
architecture PARITY_STRUCTURAL of PARITY is
  component XOR_GATE --component declaration
    port(X,Y: in BIT; Z: out BIT);
  end component;
  component INV      --component declaration
    generic(DEL: TIME);
    port(X: in BIT; Z: out BIT);
  end component;
  signal T1, T2, T3: BIT;
begin
  -- component instantianon statements:
  XOR1: XOR_GATE port map (V(0), V(1), T1);
  XOR2: XOR_GATE port map (V(2), V(3), T2);
  XOR3: XOR_GATE port map (T1, T2, T3);
  INV1: INV
    generic map (0.5 ns)
    port map (T3, EVEN);
end PARITY_STRUCTURAL;
```

Component Declaration and Instantiation

- *Component declarations* introduce templates for building blocks (sub-components) that will be used inside the architecture.
- A *component instantiation statement* creates an instance of a declared component.
  - The *port map* specifies the actual interconnections on the ports of the sub-components.
  - The *generic map* specifies actual values for the generic parameters.
- Once instantiated, components become active and work in parallel.

Component Configuration

- Component instantiation statements activate a certain architecture body related to a certain entity declaration.

(entity declaration/architecture body) pairs have to be associated to component instances.
This binding is called *component configuration*.

- *Default binding* solves configuration of the components in absence of any explicit binding indication (this has been used in the example before):
  - That entity declaration will be associated to an instance of a component which has the same name as the declared component.
  - For the association of an architecture body to the entity declaration:
    a. If one single architecture body has been defined for a given entity, that architecture will be associated.
    b. If several architecture bodies have been defined for a given entity, the most recently analyzed (compiled) will be associated.

- VHDL offers a very sophisticated mechanism to perform component configuration in a flexible manner.

A Simulation Testbench for the Example

- In order to verify a model by simulation, a testbench is usually created.

```
entity BENCH is
end BENCH;

use WORK.all;
architecture ARCH_BENCH of BENCH is
  component PARITY
    port(V: in BIT_VECTOR (3 downto 0);
         EVEN: out BIT);
  end component;
  signal VECTOR: BIT_VECTOR (3 downto 0);
  signal E: bit;
begin
  VECTOR <= ”0010”,
    ”0000” after 3 ns,
    ”1001” after 5.8 ns,
    . . .
    ”0111” after 44.5 ns,
    ”1101” after 50 ns;
  PARITY_GENERATOR:PARITY port map(VECTOR, E);
end ARCH_BENCH;
```

A Simulation Testbench (cont'd)

- The architecture body consists of two concurrent statements:
  1. A *concurrent signal assignment*.
  2. A *component instantiation*.

- Which model will actually be simulated?

  The testbench above uses the default binding

  The entity PARITY (see slide 7) will be used. But which of the two architecture bodies will be associated to it: PARITY_STRUCTURAL or PARITY_BEHAVIORAL?

  According to default binding, the one will be simulated which has been compiled most recently.

- Of course, we want to simulate any of the two models, regardless when they have been compiled!

Configuration Specification

- By a configuration specification we explicitly specify which entity declaration and architecture body to use for a certain instantiated component.

```
entity BENCH is
end BENCH;

use WORK.all;
architecture ARCH_BENCH of BENCH is
  component PARITY
    port(V: in BIT_VECTOR (3 downto 0);
         EVEN: out BIT);
  end component;
  for PARITY_GENERATOR:PARITY use
    entity PARITY(PARITY_STRUCTURAL);
  signal VECTOR: BIT_VECTOR (3 downto 0);
  signal E: bit;
begin
  VECTOR <= ”0010”,
    ”0000” after 3 ns,
    ”1001” after 5.8 ns,
    . . .
    ”0111” after 44.5 ns,
    ”1101” after 50 ns;
  PARITY_GENERATOR:PARITY port map(VECTOR, E);
end ARCH_BENCH;
```

Concurrent Statements

- The statement part of an architecture body consists of several *concurrent statements*.

  After activation of the architecture body all the concurrent statements are started and executed in *parallel* (and in parallel with the concurrent statements in all other architecture bodies which are part of the model).

Concurrent statements

- Component instantiation
- Process statement

- Concurrent signal assignment
- Concurrent procedure call
- Concurrent assertion statement

- The last three are simple short-hand notations equivalent to processes containing only a signal assignment, a procedure call, or an assertion statement respectively, together with a wait statement.

Process Statement

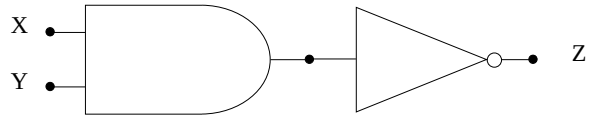
- The statement body of a process consists of a sequence of (sequential) statements which are executed one after the other (see slide 8).

- The process is an implicit loop.

- After being created at the start of the simulation, the process is either in an active state or is suspended and waiting for a certain event to occur.

  Suspension of a process results after execution of a wait statement. This wait statement can be:
  - implicit
  - explicitly specified by the designer.

Process Statement (cont'd)




```
entity AND_WITH_NOT is
 port(X, Y:in BIT;
    Z: out BIT);
end AND_WITH_NOT;

architecture SIMPLE_1 of AND_WITH_NOT is
 signal S: BIT;
begin
 AND_GATE: process
 begin
  S<=X and Y after 1 ns;
  wait on X,Y;
 end process;
 INVERTER: process
 begin
  Z<=not S after 0.5 ns;
  wait on S;
 end process;
end SIMPLE_1;
```


Process Statement (cont'd)

- If the process has a *sensitivity list*, a wait statement is automatically introduced at the end of the statement list.

The following is equivalent with the specification before:

```
entity AND_WITH_NOT is
 port(X, Y:in BIT;
    Z: out BIT);
end AND_WITH_NOT;

architecture SIMPLE_2 of AND_WITH_NOT is
 signal S: BIT;
begin
 AND_GATE: process(X,Y)
 begin
  S<=X and Y after 1 ns;
 end process;
 INVERTER: process(S)
 begin
  Z<=not S after 0.5 ns;
 end process;
end SIMPLE_2;
```

The wait statement

- A process may suspend itself by executing a wait statement:

```
wait on A,B,C until A<2*B for 100 ns;
```

  - Sensitivity clause (list of signals)
  - Condition clause
  - Time-out clause

Signals

- A VHDL object is a named entity that has a value of a given type.

  Objects in VHDL: constants, *signals*, variables, files.

- Signals are used to connect different parts of the design.

  Signals are the objects through which information is propagated between processes and between subcomponents of an entity.

  Ports are implicitly objects of class signal.

  A signal declaration is similar to the declaration of a variable. Signals may not be declared within processes or subprograms.

- The semantics of signals is closely connected to the notion of time in VHDL:
  A signal has not only a current value but also a projected waveform with determines its future values at certain moments of simulation time.

Concurrent Signal Assignment

- A signal assignment that appears as part of an architecture body (outside a process or a subprogram) is interpreted as a concurrent statement.

  Such a concurrent signal assignment is equivalent to a process containing only that particular signal assignment followed by a wait statement.
  The wait is on the signals occurring in the expression on the right side of the assignment.

The following is equivalent to the models on slides 20, 21:

```
entity AND_WITH_NOT is
 port(X, Y:in BIT;
   Z: out BIT);
end AND_WITH_NOT;

architecture SIMPLE_3 of AND_WITH_NOT is
 signal S: BIT;
begin
  S<=X and Y after 1 ns;
  Z<=not S after 0.5 ns;
end SIMPLE_3;
```

Such a (behavioral) model nicely reflects the dataflow through the design.

The VHDL Simulation Mechanism

- After *elaboration* of a VHDL model results a set of processes connected through signals.
- The VHDL model is simulated under control of an event driven simulation kernel (*the VHDL simulator*).
- Simulation is a cyclic process; each *simulation cycle* consists of a *signal update* and a *process execution* phase.
- A global clock holds the *current simulation time*; as part of the simulation cycle this clock is incremented with discrete values.

The VHDL Simulation Mechanism (cont'd)

Essential feature:

current signal values are *only* updated **by the simulator at certain moments during simulation**!

```
...
X<=1;
if X=1 then
 statement_sequence_1
else
 statement_sequence_2
end if;
...
```

- A signal assignment statement only schedules a new value to be placed on the signal at some later time which is specified by the designer as part of the signal assignment:

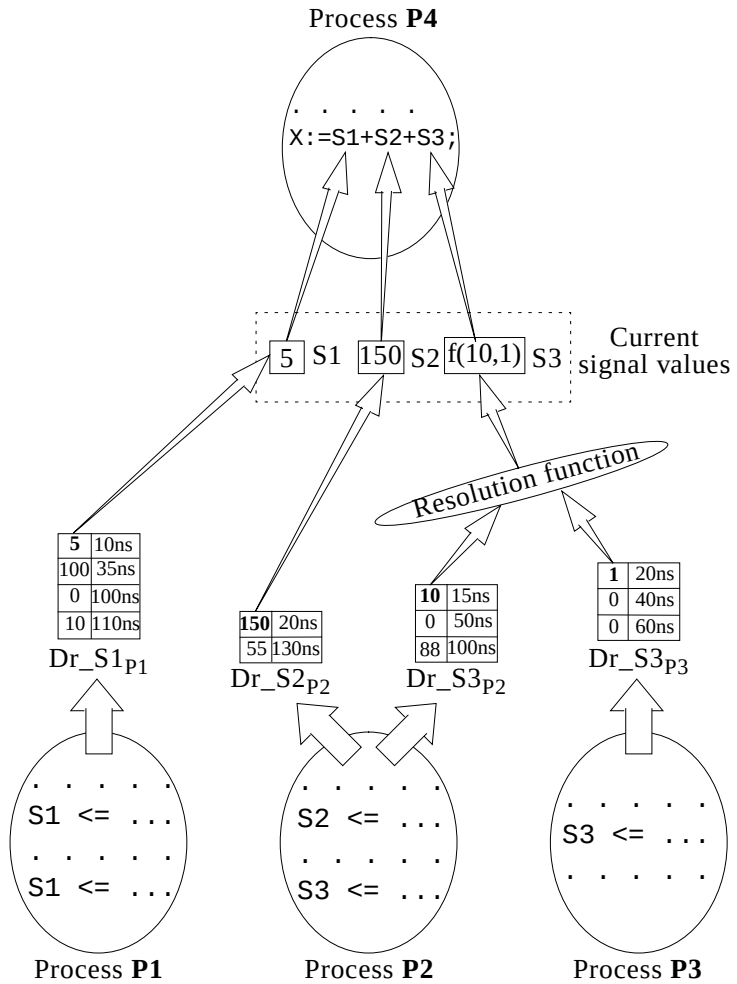
```
S<=1 after 20 ns,15 after 35 ns;
```

The VHDL Simulation Mechanism (cont'd)

- signal driver contains the *projected output waveform* of a signal;
  a process that assigns values to a signal will automatically create a driver for that signal;
- projected output waveform is a set of *transactions*;
- transaction: pair consisting of a *value* and a *time*.

A signal assignment only affects the projected output waveform, by placing one or more transactions into the driver corresponding to the signal and possibly by deleting other transactions.

The VHDL Simulation Mechanism (cont'd)



The VHDL Simulation Mechanism (cont'd)

- As simulation time advances and the current time becomes equal to the time component of the next transaction, the first transaction is deleted and the next becomes the *current value of the driver*.

  ⇩

  The driver gets a new value.

  Regardless if this value is different from the previous one or not, the driver and the signal is said to be *active* during that simulation cycle.

- During each simulation cycle, the *current value of the signal* is updated for those signals which have been active during that cycle.

  ⇩

  If, as result, the current value of the signal has changed, an *event* has occurred on that signal.

- *Resolved signal*: a signal for which several drivers exist (several processes assign values to that signal). For each resolved signal the designer has to specify an associated *resolution function*.

The VHDL Simulation Cycle

- The current time T_c is set to T_n;
- Each active signal is updated; as result of signal updates events are generated.
- Each process that was suspended waiting on signal events that occurred in this simulation cycle resumes; processes also resume which were waiting for a certain, completed, time to elapse;
- Each resumed process executes until it suspends;
- The time T_n of the next simulation cycle is determined as the earliest of the following three time values:
  1. TIME'HIGH;
  2. The next time at which a driver becomes active
  3. The next time at which a process resumes;

Delta Delay and Delta Cycle

- The simulation philosophy of VHDL is based on the ordering of events in time:

  new events are generated as result of actions taken in response to other events scheduled for previous simulation times.

The following concurrent signal assignment statement is executed in response to an event on signal X, let's say at time *t*:

```
S<=X+1 after 20 ns,X+15 after 35 ns;
```

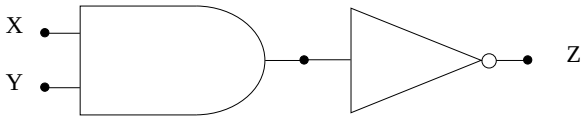
In response, two events will be planned on signal S, for times *t*+20 and *t*+35, respectively.

- What if, in response to an event at time *t*, another event at the same time is generated?

```
S<=X+1;
```

- Different events that occur at the same simulation time are ordered and handled in successive simulation cycles, preserving their cause/effect relationship.

Delta Delay and Delta Cycle (cont'd)




```
entity DELTA_DELAY_EXAMPLE is
 port(X, Y:in BIT;
    Z: out BIT);
end DELTA_DELAY_EXAMPLE;

architecture DELTA of DELTA_DELAY_EXAMPLE
is
 signal S: BIT;
begin
 AND_GATE: process(X,Y)
 begin
  S<=X and Y;
 end process;
 INVERTER: process(S)
 begin
  Z<=not S;
 end process;
end DELTA;
```

Delta Delay and Delta Cycle (cont'd)

- If the previous model is simulated, successive simulation cycles will be executed at the same simulation time.
  Such cycles are separated by a so called *delta-delay*.

A delta-delay is an infinitesimally small delay that separates events occurring in successive simulation cycles but at the same simulation time.

A simulation cycle that is performed at the same simulation time as the previous one is called a *delta cycle*.

- This mechanism allows a correct simulation of models where *the delay of some components is ignored* and, thus, there is no difference in simulation time between the events on input and output of this components.

- For some models, developed for synthesis, we do not know the delays before synthesis. In the input model, delays are ignored. For such a model, functionality can be checked by simulation, but not timing.
  The model generated after synthesis, contains delays and both correct functionality and timing can be checked.

Signal Assignment Statement

The projected output waveform stored in the driver of a signal can be modified by a *signal assignment statement*.

signal_assignment_statement ::=
target <= [**transport** | [**reject** *time*_expression] **inertial**] waveform;

waveform ::=
waveform_element {, waveform_element}

waveform_element ::=
*value*_expression [**after** *time*_expression]

```
S<=transport 100 after 20 ns, 15 after 35 ns;
S <= 1 after 20 ns,15 after 35 ns;
```

- The concrete way a driver is updated as result of a signal assignment, depends on the *delay mechanism* (*transport* or *inertial*).
- The delay mechanism can be explicitly specified as part of the signal assignment; if no mechanism is specified, the default is *inertial*.

Transport Delay

Transport delay models devices that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration.

This is typical when modeling transmission lines.

☞ No transaction scheduled to be executed *before* a new one is affected by a signal assignment with transport delay.

Update rule:

1. All old transactions scheduled to occur at the same time or after the first new transaction are deleted from the projected waveform.
2. The new transactions are appended to the end of the driver.

Transport Delay (cont'd)

Examples

Consider the following assignments executed at simulation time 100 ns (the projected waveform, at that moment, consists of a single transaction with value 0):

```
S<=transport 100 after 20 ns, 15 after 35 ns;
S<=transport 10 after 40 ns;
S<=transport 25 after 38 ns;
```

Driver for *S* after first two assignments:

0	100	15	10
100 ns	120 ns	135 ns	140 ns

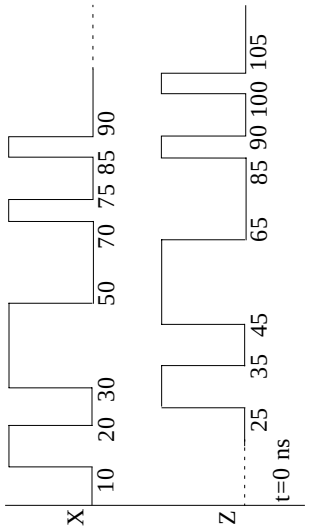
Driver for *S* after last assignment:

0	100	15	25
100 ns	120 ns	135 ns	138 ns

- Every change on the input will be processed, regardless of how short the time interval between this change and the next one.

Transport Delay (cont'd)

A buffer element with delay 15 ns — transport delay



Inertial Delay

Inertial delay models the timing behavior of current switching circuits: an input value must be stable for a certain duration, called *pulse rejection limit*, before the value propagates to the output.

```
S <= reject 5 ns inertial X after 10 ns;
```

- Additional update rule (after update operations have been performed exactly like for transport delay):

  All old transactions scheduled to occur at times between the time of the first new transaction and this time minus the pulse rejection limit are deleted from the projected waveform; excepted are those transactions which are immediately preceding the first new transaction and have the same value with it.

Inertial Delay (cont'd)

- If no pulse rejection limit is specified, it is considered to be equal with the time value in the first waveform element

```
S <= X after 10 ns, 0 after 25 ns;
```

is equivalent to:

```
S <= reject 10 ns inertial X after 10 ns,
     0 after 25 ns;
```

Inertial Delay (cont'd)

Examples

Consider the assignments below, executed at simulation time 100 ns, when the driver for signal S has the following contents:

0	1	15
100 ns	110 ns	135 ns

```
S <= 8 after 20 ns,2 after 40 ns,
          5 after 65 ns,10 after 100 ns;
S <= reject 55 ns inertial 5 after 90 ns;
```

Driver for S after first assignment:

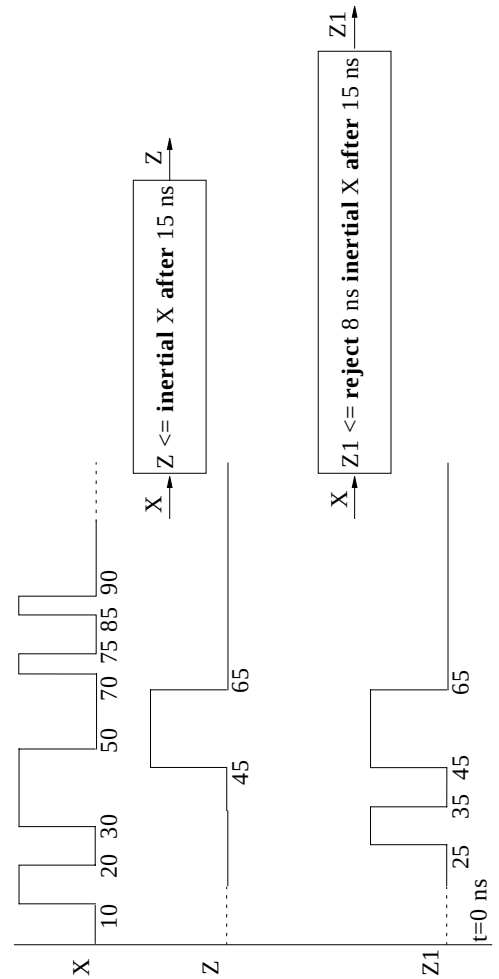
0	8	2	5	10
100 ns	120 ns	140 ns	165 ns	200 ns

Driver for S after second assignment:

0	8	5	5
100 ns	120 ns	165 ns	190 ns

Inertial Delay (canto)

A buffer element with delay 15 ns — inertial



Resolved Signals and Resolution Functions

- *Resolved signal*: a signal for which several drivers exist (several processes assign values to that signal). For each resolved signal the designer has to specify an associated *resolution function*.
- The resolution function computes the value which is used to update the current signal value, depending on the actual values of the drivers.
- The resolution function is automatically called by the simulation kernel every time the signal value has to be updated.

Resolved Signals and Resolution Functions (cont'd)

Example:

A resolved signal, `Line`, which models an interconnection line to which the output of several devices is connected. Each device is modeled by one process.

The resolution function implements a *wired or*.

```
architecture Example of ... is
  type Bit4 is ('X','0','1','Z');
  type B_Vector is array(Integer range <>)
                                    of Bit4;

  function Wired_Or(Input: B_Vector)
                               return Bit4 is
    variable Result: Bit4:='0';
  begin
    for I in Input'Range loop
     if Input(I)='1' then
        Result:='1';
        exit;
     elsif Input(I)='X' then
        Result:='X';
     end if;
    end loop;
    return Result;
  end Wired_or;
```

Example (cont'd)

```
  signal Line: Wired_Or Bit4;

begin
  P1: process
  begin
    - - - - - -
    Line <= '1';
    - - - - - -
  end process;

  P2: process
  begin
    - - - - - -
    Line <= '0';
    - - - - - -
  end process;
end Example.
```

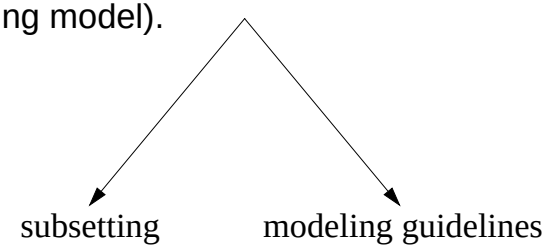
- Each time a resolution function is invoked by the simulation kernel, it is passed an array value, each element of which is determined by a driver of the corresponding resolved signal.

VHDL For System Synthesis

- Semantic of VHDL is simulation based
- VHDL widely used for synthesis

Problems:

1. VHDL has the rich capabilities of a modern programming language $\Rightarrow$ some facilities are not relevant for hardware synthesis.
2. Some features are semantically explained in terms of simulation (process interaction, timing model).



VHDL For System Synthesis (cont'd)

- VHDL - Synthesis tools at logic and RT level are commonly available today. IEEE standards will be released soon for interpretation and use of VHDL in logic synthesis.

- Industrial use of high-level synthesis with VHDL is at the beginning.

VHDL For System Synthesis (cont'd)

Commonly accepted restrictions for High-Level synthesis:

1. Features which are not relevant from HL synthesis point of view are excluded: structural specification, resolution functions, certain signal attributes, etc.
2. The input specification is purely sequential - formulated as a single VHDL process.
3. All synchronization is restricted to a clock signal; wait statements are allowed exclusively on this explicit clock. Signal assignment are allowed only on output ports.
4. The scheduling of certain operations is fixed in terms of clock cycles.
5. VHDL strict timing is not considered.

not acceptable in the context of system-level synthesis