Specification and Simulation of Combinational Designs using VHDL

With thanks to Sudhakar Yalamanchili

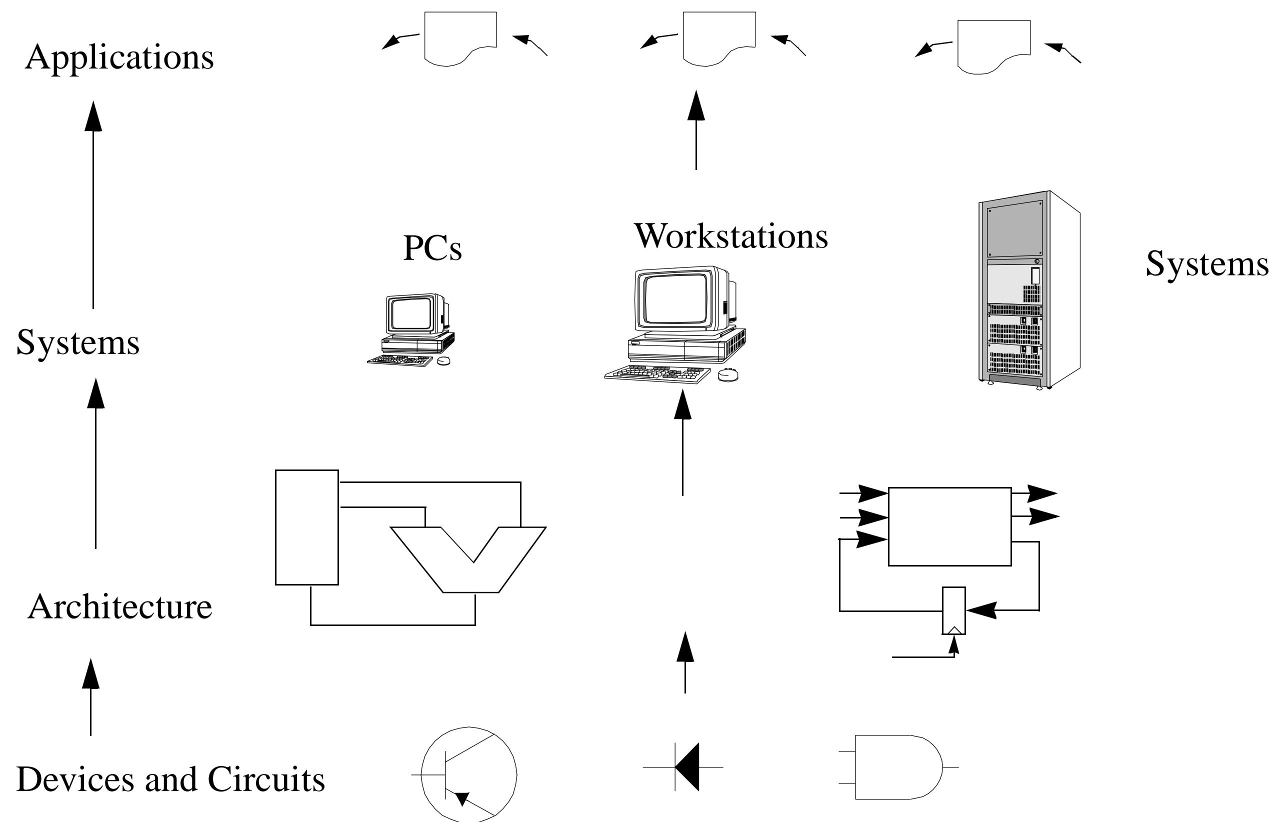
Overview

1. **Introduction to VHDL**
   – Use in describing systems
2. **Describing & modelling systems**
   – Interface & behaviour
   – Signals, events, timing, concurrency
   – Use of discrete event simulation
3. **Basic language constructs**
   – Std_logic resolved type
   – Signal assignments
   – Delays
   – Modelling complex behaviour
   – Modelling structure
4. **Use of VHDL to model and simulate sequential logic designs**

1. Introduction to VHDL

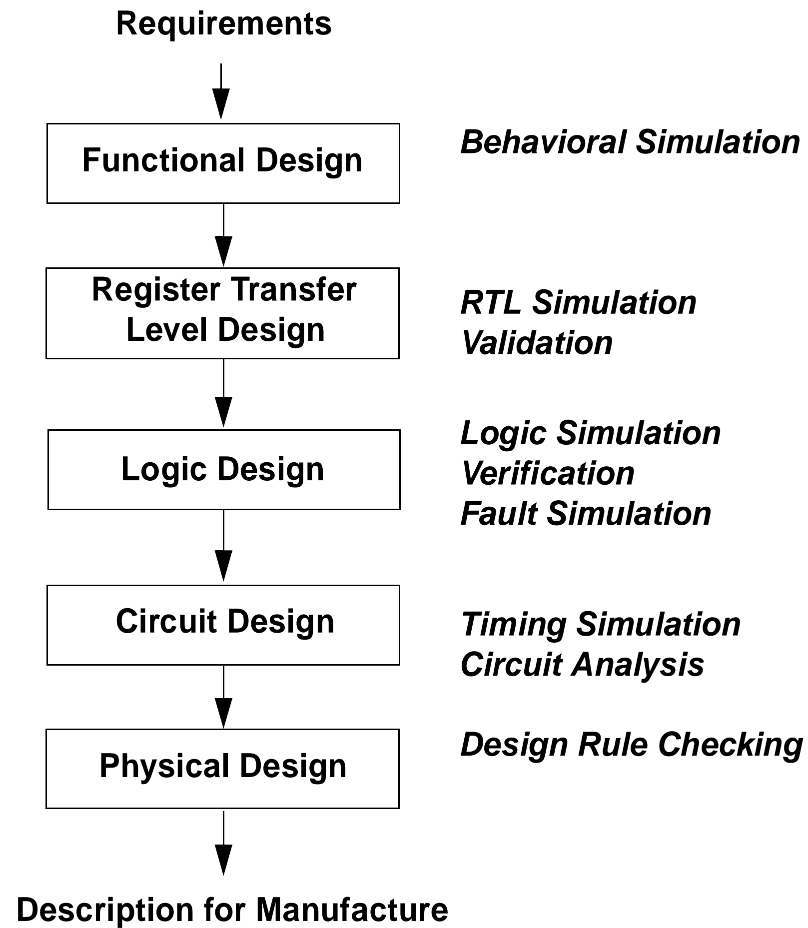
- **Use in describing systems**

Describing Digital Systems



- Systems may be described at multiple levels of abstraction

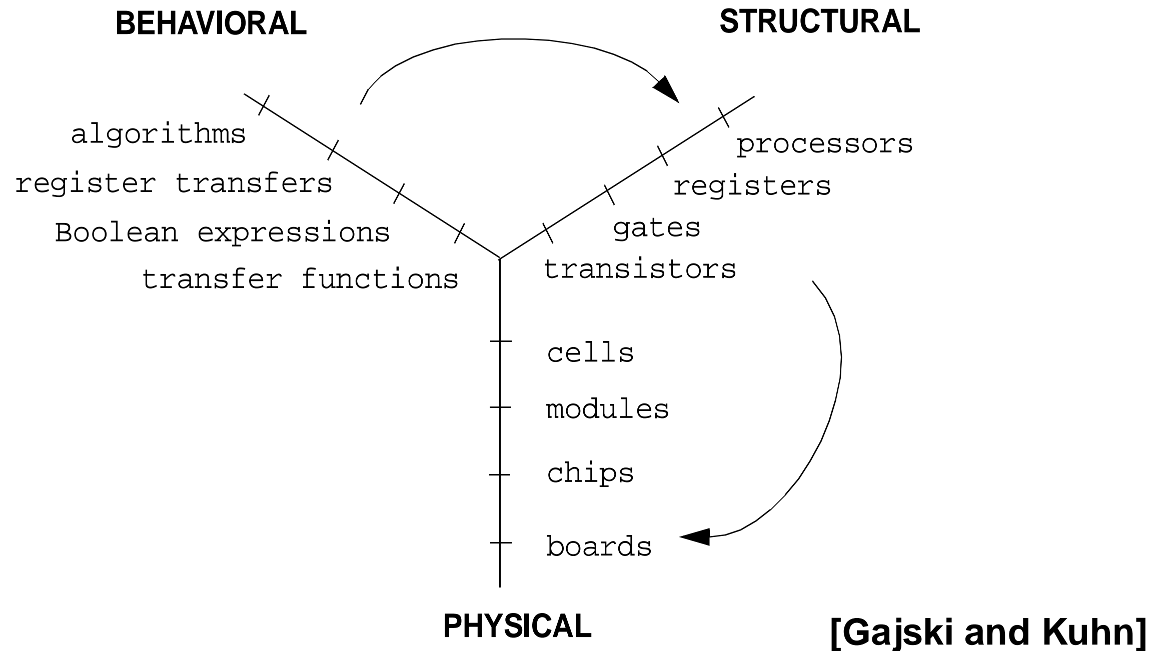
The Digital System Design Process



Why Do We Describe Systems?

- Design Specification
  - unambiguous definition of components and interfaces in a large design
- Design Simulation
  - verify system/subsystem/chip performance prior to design implementation
- Design Synthesis
  - automated generation of a hardware design

Hardware Description Languages



[Gajski and Kuhn]

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction

The VHDL Language

V *Very High Speed Integrated Circuit*

H *Hardware*

D *Description*

L *Language*

- Interoperability between design tools: standardized portable model of electronic systems
- Technology independent description
- Reuse of components described in VHDL

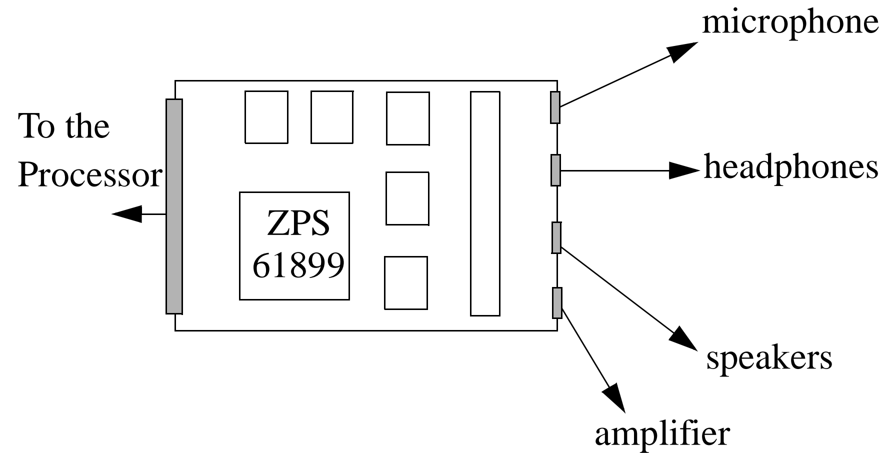
History of VHDL

- Designed by IBM, Texas Instruments, and Intermetrics as part of the DoD funded VHSIC program
- Standardized by the IEEE in 1987: IEEE 1076-1987
- Enhanced version of the language defined in 1993: IEEE 1076-1993
- Additional standardized packages provide definitions of data types and expressions of timing data
    - IEEE 1164 (data types)
    - IEEE 1076.3 (numeric)
    - IEEE 1076.4 (timing)

2. Describing & modelling systems

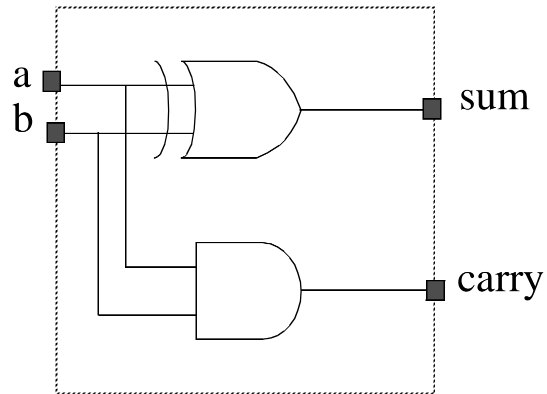
- Interface & behaviour
- Signals, events, timing, concurrency
- Use of discrete event simulation

Describing Systems



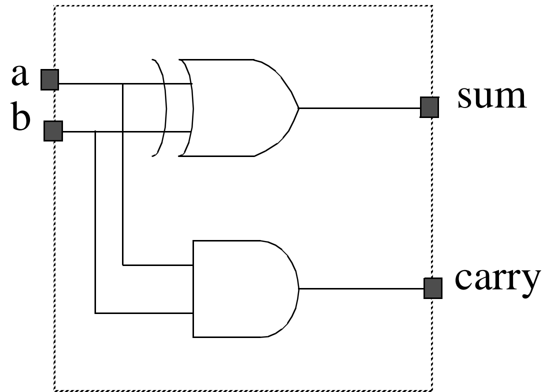
- System: “An assemblage of objects united by some form or regular interaction or dependence”
- What aspects of a digital system do we want to describe?
  - interface
  - behavior: functional and structural

Basic Language Concepts



- What aspects of a digital system do we want to describe?
    - interface: how do we connect to it
    - behavior: what does it do?

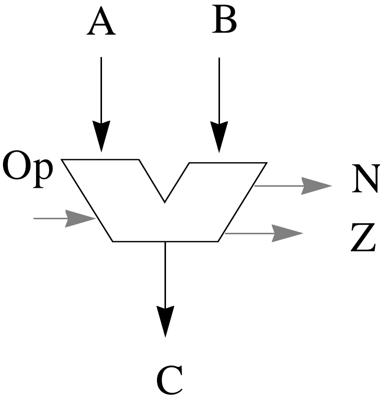
Describing the Interface: The Entity Construct



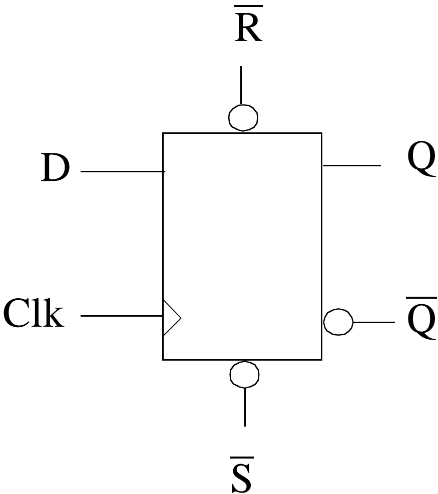

```
entity half_adder is
port ( a, b : in bit;
       sum, carry :out bit);
end half_adder;
```

- The interface is a collection of *ports*
  - ports are a new programming object: *signal*
  - ports have a type, e.g., **bit**
  - ports have a mode: in, out, inout (bidirectional)

Example Entity Descriptions

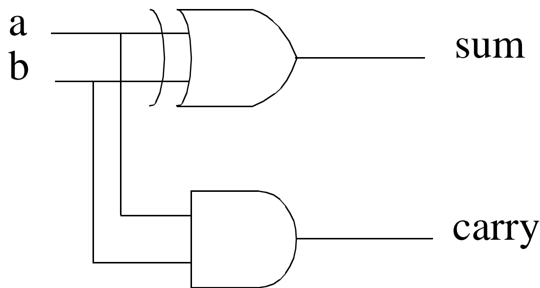


entity `ALU32` **is**
port(`A`, `B` **: in bit_vector** (31 **downto** 0);
`C` : **out bit_vector** (31 **downto** 0);
`Op` **: in bit_vector** (5 **downto** 0);
`N`, `Z` **: out bit**);
end `ALU32`;



entity `D_ff` **is**
port(`D`, `Q`, `Clk`, `R`, `S`: **in bit**;
`Q`, `Qbar` : **out bit**);
end `D_ff`;

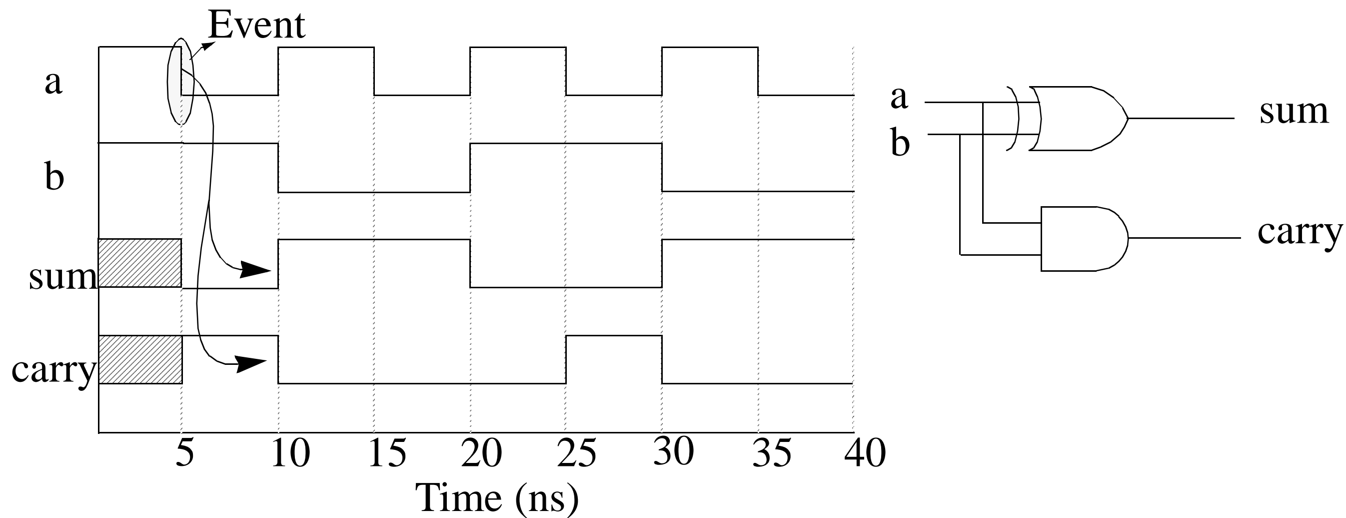
Describing Behavior: The Architecture Construct




```
entity half_adder is
port (a, b : in bit;
        sum, carry :out bit);
end  half_adder;

architecture behavioral of half_adder
is
begin
sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end behavior;
```

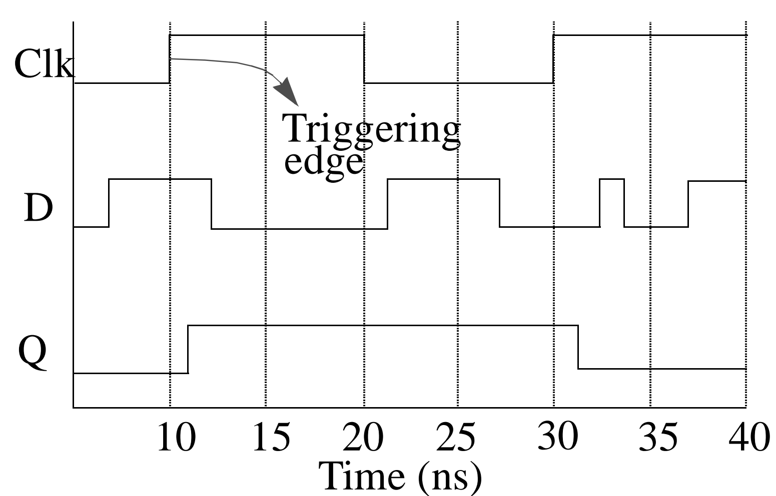
- Description of events on output signals in terms of events on input signals: the *signal assignment statement*
- Specification of propagation delays
- Type **bit** is not powerful enough for realistic simulation: use the IEEE 1164 value system

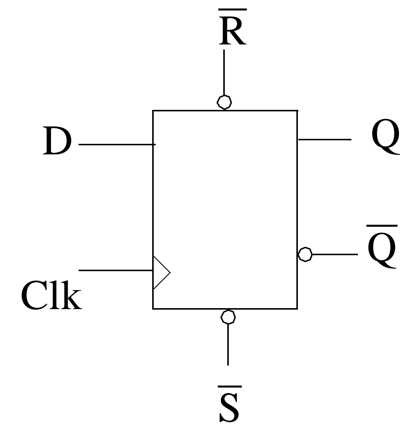
Behavioral Attributes of Digital Systems



- Digital systems are about *signals* and their *values*
- *Events*, *propagation delays*, *concurrency*
- Time ordered sequence of events produces a *waveform*

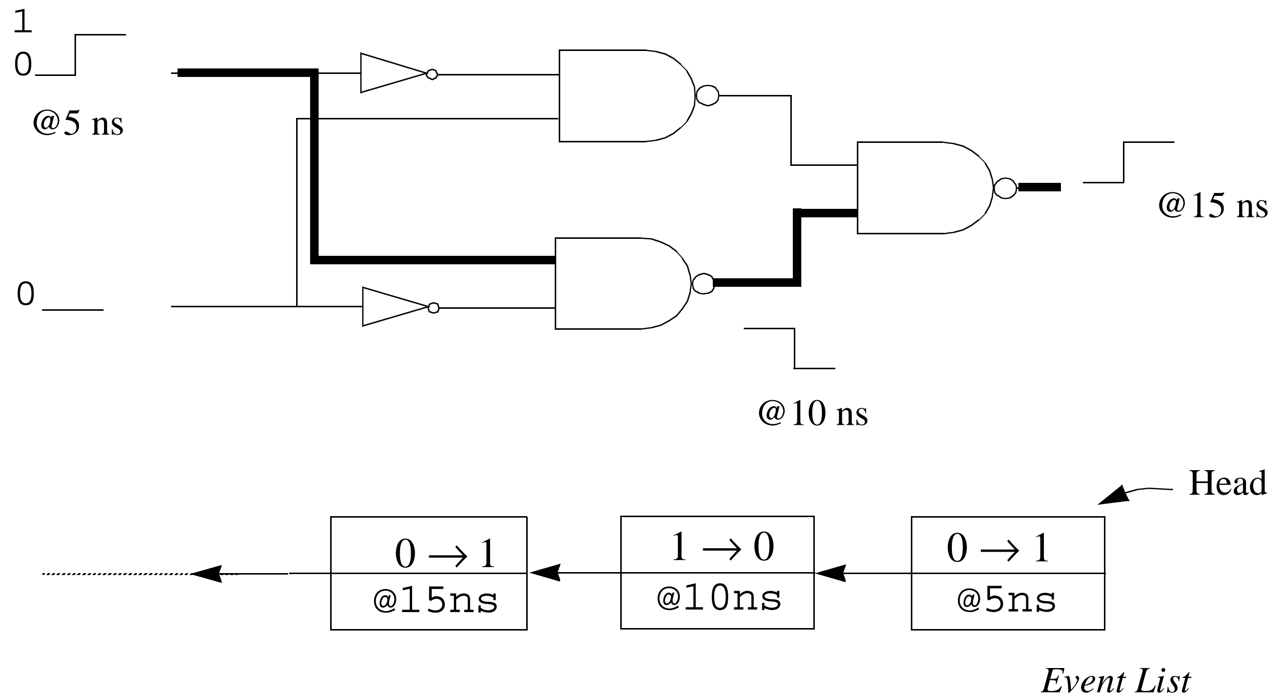
Behavioral Attributes of Digital Systems





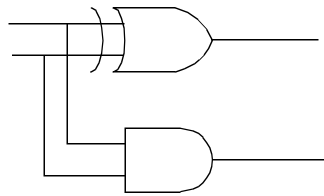
- *Timing*: computation of events takes place at specific points in time
- Timing is an attribute of both synchronous and asynchronous systems

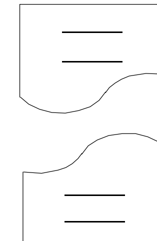
Simulation of Digital Systems



- Digital systems propagate events
- Discrete event simulations manage the generation and recording of events

Simulation Modeling



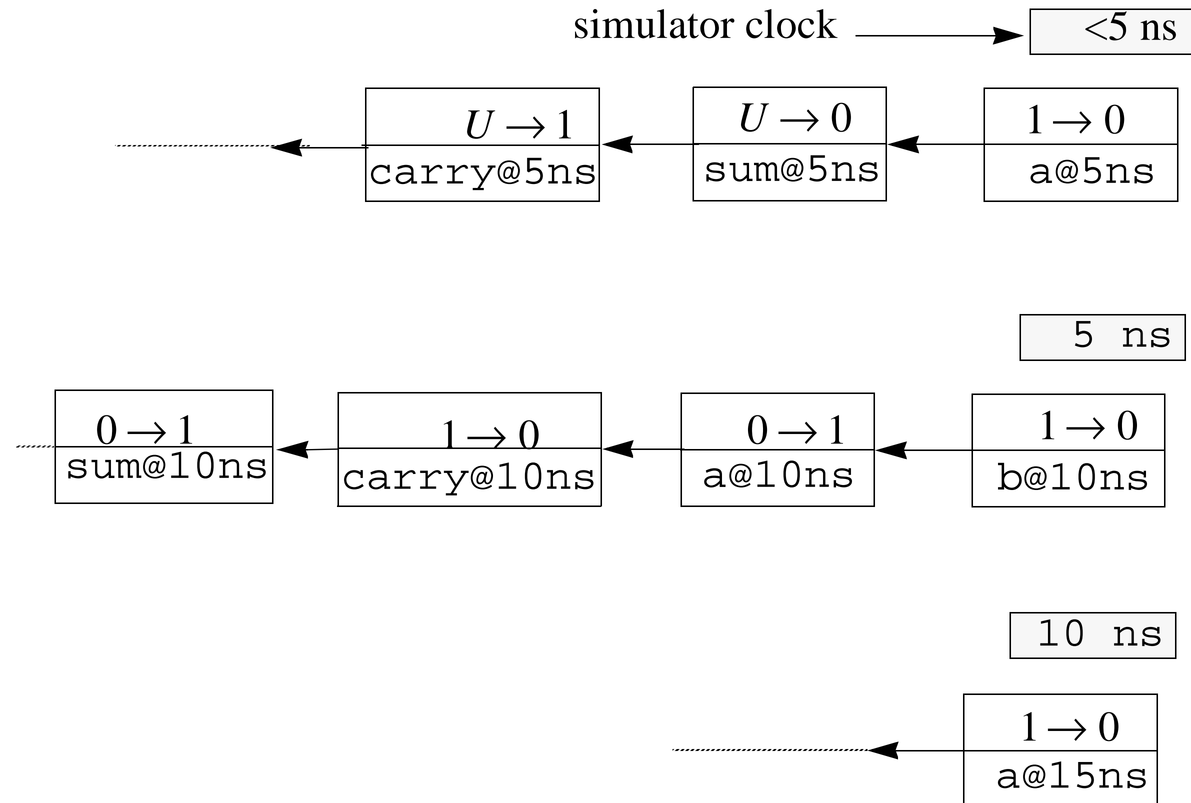


HDL Description

Altera, Viewlogic, Xilinx

- VHDL programs describe the generation of events in digital systems
- Discrete event simulator manages event ordering and progression of time

Discrete Event Simulation: Half Adder Model



- Management of simulation time: ordering of events
- Two step model of the progression of time

Discrete Event Simulation Algorithm

repeat until event_list empty or preset simulation time expired:

1. **Advance simulation time to that of the next event (earliest timestamp in the event list) and process all events (update all signals receiving values) at this time**
2. **Evaluate all components affected by signal updates and schedule any future events generated by inserting them into the event list**

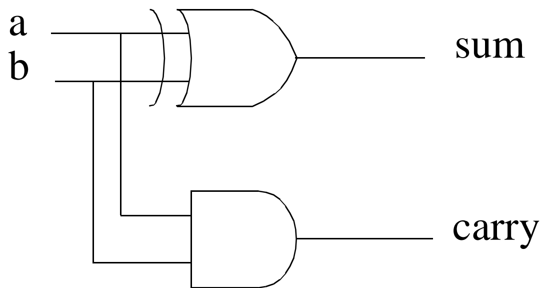
Modeling Digital Systems

- VHDL programs describe behavioral attributes of digital systems
  - events, propagation delays, concurrency
  - waveforms and timing
  - signal values
- Discrete event simulators mimic the operation of the system being described
  - two step model of time
    - assign values scheduled for signals at the current time
    - process affected components and schedule future signal values
  - ensures events are generated and processed in the correct order
  - a correct simulation generates *only and all* those events that would have been generated in the physical system

3. Basic language constructs

- **Std_logic resolved type**
- **Signal assignments**
- **Delays**
- **Modelling complex behaviour**
- **Modelling structure**

Describing Behavior: The Architecture Construct

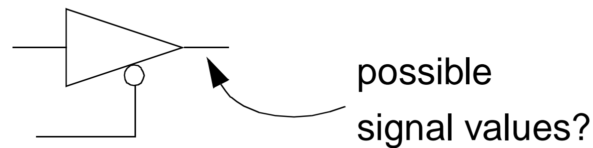



```
entity half_adder is
port (a, b : in bit;
        sum, carry :out bit);
end half_adder;

architecture behavioral of half_adder
is
begin
sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end behavior;
```

- Description of events on output signals in terms of events on input signals: the *signal assignment statement*
- Specification of propagation delays
- Type **bit** is not powerful enough for realistic simulation: use the IEEE 1164 value system

Behavioral Attributes of Digital Systems

- We associate logical values with the state of a signal



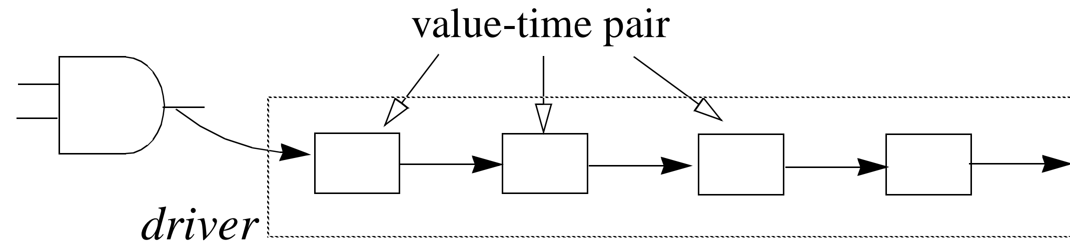
possible
signal values?

- Signal Values: IEEE 1164 Value System

Value	Interpretation
U	Uninitialized
X	Forcing Unknown
0	Forcing 0
1	Forcing 1
Z	High Impedance
W	Weak Unknown
L	Weak 0
H	Weak 1
-	Don't Care

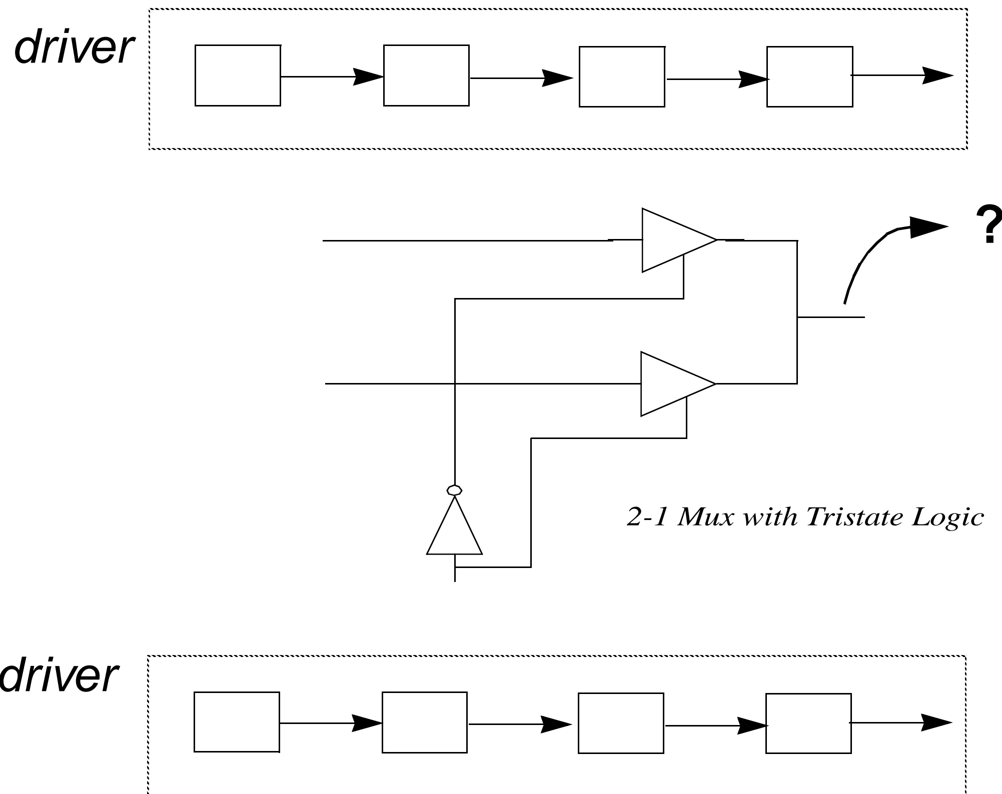
Implementation of Signals

- The structure of a signal assignment statement
    - *signal* <= (*value expression* **after** *time expression*)
    - RHS is referred to as a *waveform element*
- Every signal has associated with it a *driver*



- holds the current and future values of the signal - a *projected waveform*
- signal assignment statements modify the driver of a signal
- value of a signal is the value at the head of the driver
- note the hardware analogy

Shared Signals



2-1 Mux with Tristate Logic

- How do we model the state of a wire?
- Rules for determining the signal value is captured in the resolution function

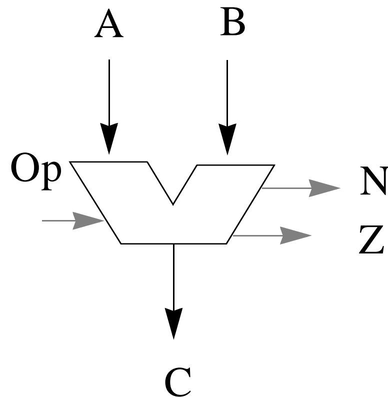
Resolution Function: std_logic & resolved()

- resolving values for std_logic types

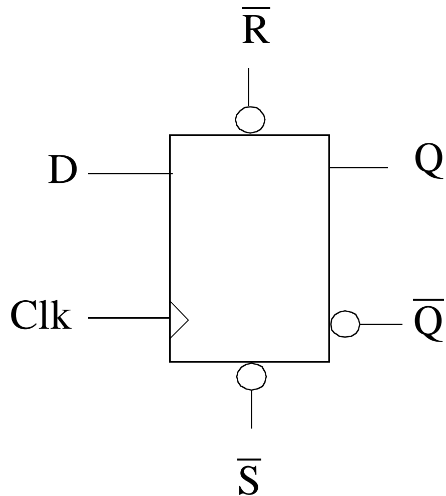
	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

- Pairwise resolution of signal values from multiple drivers
- Resolution operation must be associative

Example Entity Descriptions: IEEE 1164

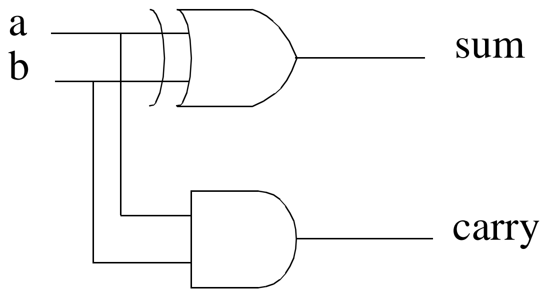



```
entity ALU32 is
port(  A, B: in std_logic_vector  (31 downto 0);
       C : out std_logic_vector  (31 downto 0);
       Op: in std_logic_vector  (5 downto 0);
       N, Z: out std_logic);
end ALU32;
```




```
entity D_ff is
port(  D, Q, Clk, R, S: in std_ulogic;
       Q, Qbar : out std_ulogic);
end D_ff;
```

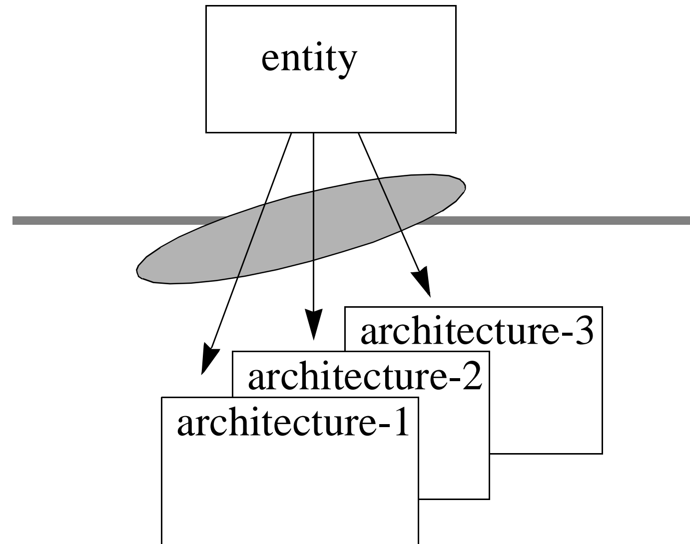
Describing Behavior: The Architecture Construct




```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port ( a, b : in std_logic;
        sum, carry :out std_logic);
end  half_adder;

architecture behavioral of half_adder
is
begin
sum <= (a xor b) after 5 ns;
carry <= (a and b) after 5 ns;
end behavior;
```

- Use of the IEEE 1164 value system simply requires inclusion of the library and package declarations statements

Design Units

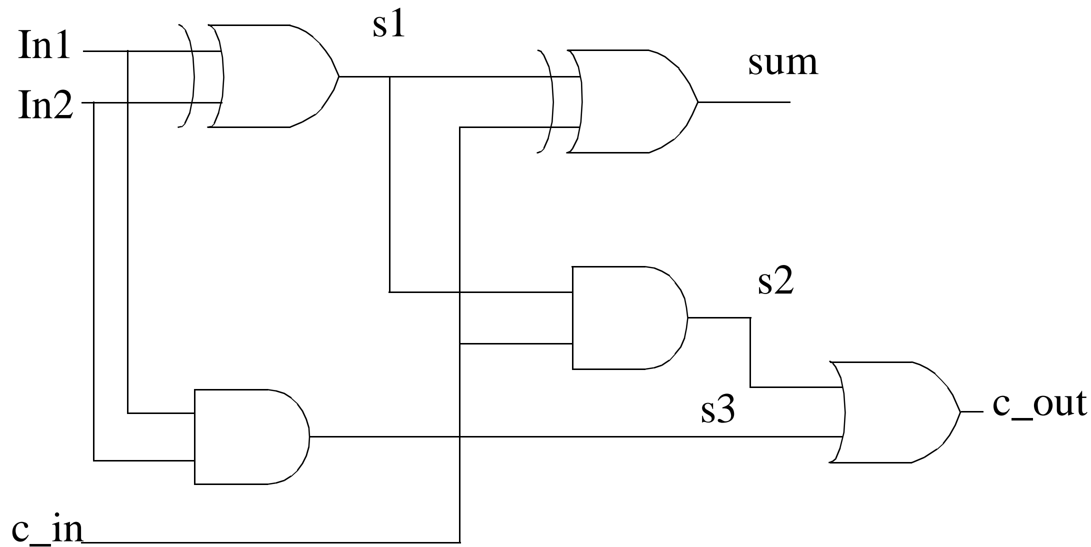


- An entity may have multiple architectures
- Separation of interface from implementation

Simple Signal Assignment

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (in1, in2, c_in: in std_logic;
      sum, c_out: out std_logic);
end full_adder;
```

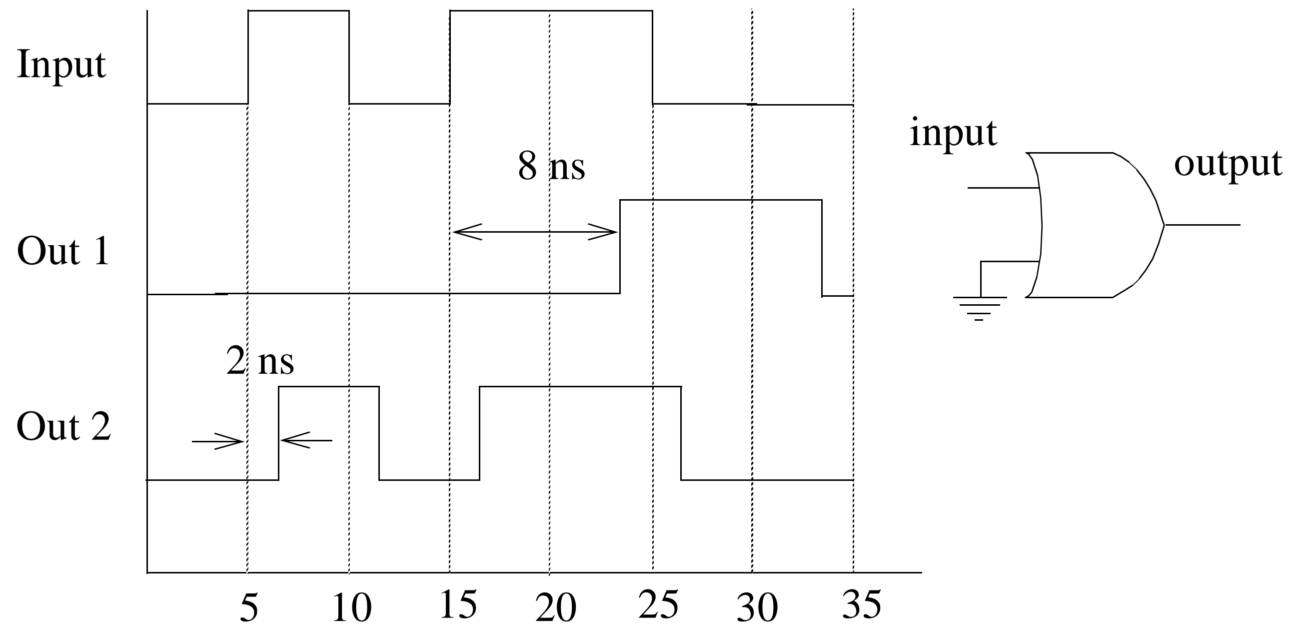
```
architecture dataflow of full_adder is
signal s1, s2, s3  : std_logic;
constant gate_delay: Time:= 5 ns;
begin
L1:  s1 <= (In1 xor In2) after gate_delay;
L2:  s2 <= (c_in and s1) after gate_delay;
L3:  s3 <= (In1 and In2) after gate_delay;
L4:  sum <= (s1 xor c_in) after gate_delay;
L5:  c_out <= (s2 or s3) after gate_delay;
end dataflow;
```



Simple Signal Assignment Statement

- The *constant* programming object
  - values cannot be changed
- Use of signals in the architecture
  - internal signals used to connect components
- A statement is executed when a signal in the RHS has value assigned to it
  - 1-1 correspondence between signal assignment statements and signals in the circuit
  - order of statement execution follows propagation of events in the circuit
  - textual order *does not* imply execution order
  - trace the statement execution order when In1 changes value

Understanding Delays



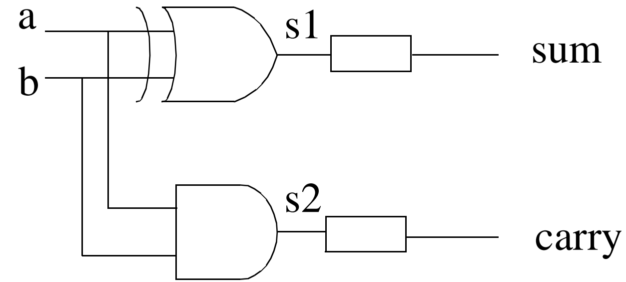
- Inertial delay model
- Transport delay model
- Delta delay model

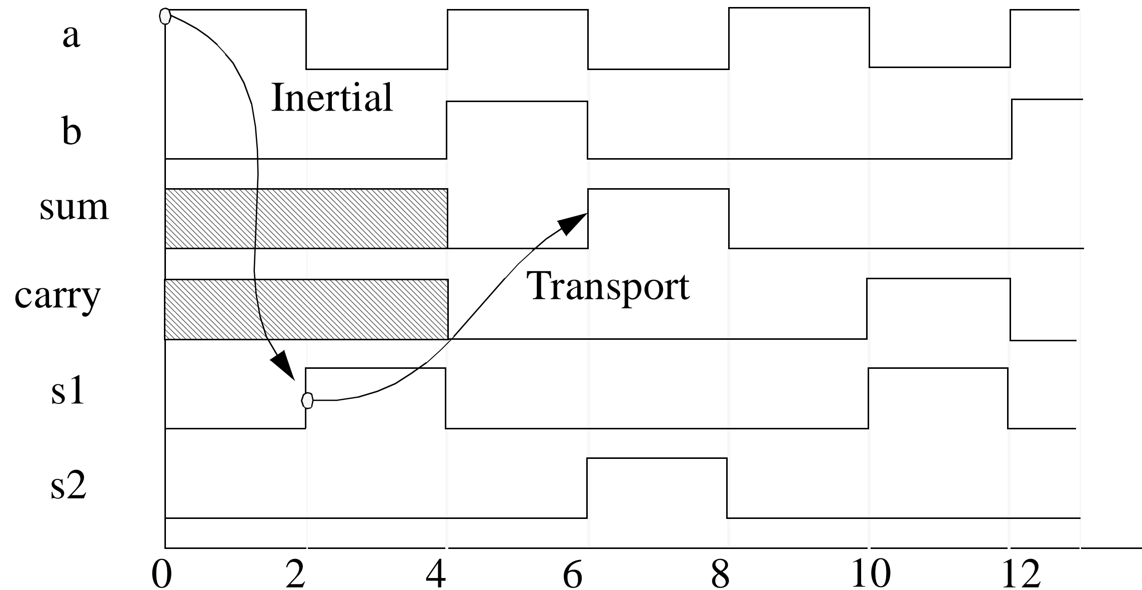
Understanding Delays

- Inertial delay
    - default delay model
    - suitable for modeling delays through devices such as gates
- Transport Delay
    - to model delays through devices with very small inertia, e.g, wires
    - all input events are propagated to output signals
- Delta delay
    - what about models where no propagation delays are specified?
    - infinitesimally small delay is automatically inserted by the simulator to preserve correct ordering of events

Transport Delays: Example

```
architecture transport_delay of
half_adder is
signal s1, s2: std_logic:= '0';
begin
s1 <= (a xor b) after 2 ns;
s2 <= (a and b) after 2 ns;
sum <= transport  s1 after 4 ns;
carry <= transport  s2 after 4 ns;
end transport_delay;
```

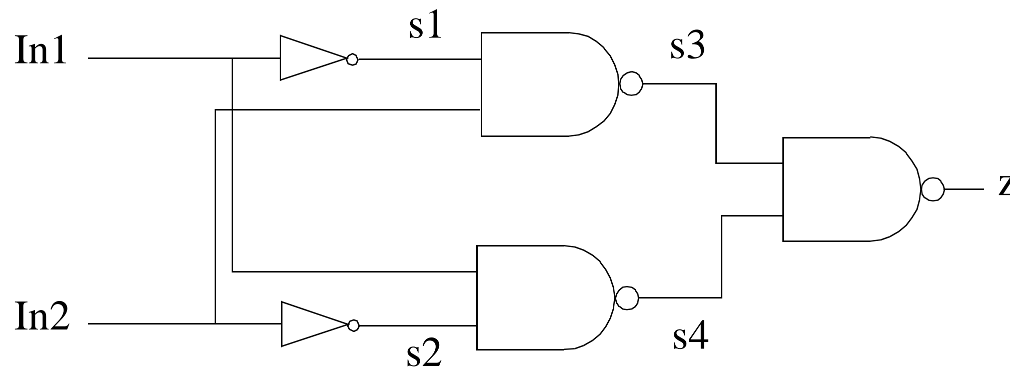




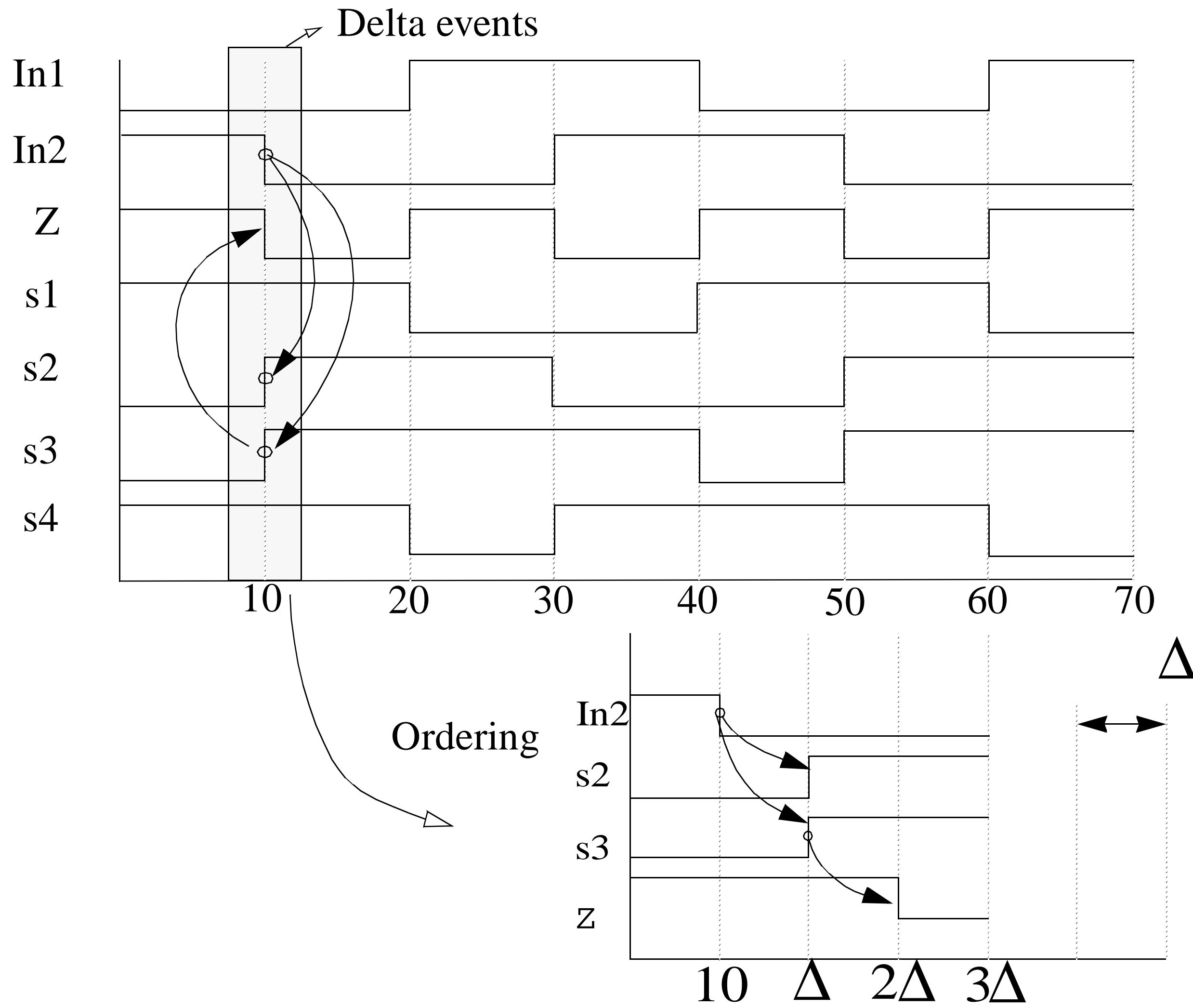
Delta Delays: Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity combinational is
port (In1, In2: in std_logic;
     z  : out std_logic);
end combinational;
```

```
architecture behavior of
combinational is
signal s1, s2, s3, s4: std_logic:=
begin
s1 <= not  In1;
s2 <= not In2;
s3 <= not (s1 and In2);
s4 <= not (s2 and In1);
z <= not (s3 and s4);
end behavior;
```

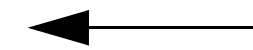


Delta Delays: Example



Conditional Signal Assignment

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (   In0, In1, In2, In3  : in std_logic_vector  (7 downto 0);
         Sel: in std_logic_vector(1  downto  0);
         Z  : out std_logic_vector  (7 downto 0));
end mux4;

architecture behavioral of mux4 is
begin
Z <= In0  after 5 ns when Sel = "00" else
     In1  after 5 ns when Sel = "01" else
     In2  after 5 ns when Sel = "10" else
     In3  after 5 ns when Sel = "11" else
     "00000000" after 5 ns;
end behavioral;
```

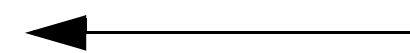


Evaluation order is important!

- First true conditional determines the output value

Selected Signal Assignment Statement

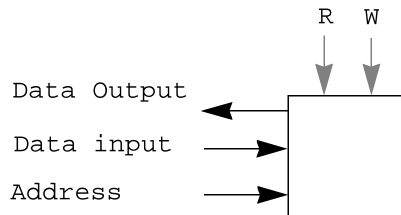
```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (   In0, In1, In2, In3  : in std_logic_vector  (7 downto 0);
         Sel: in std_logic_vector(1  downto  0);
         Z  : out std_logic_vector  (7 downto 0));
end mux4;

architecture behavioral-2 of mux4 is
begin
with Sel select
Z <= (In0 after 5 ns) when "00",
     (In1 after 5 ns) when "01",
     (In2 after 5 ns) when "10",
     (In3 after 5 ns) when "11"
     (In3 after 5 ns) when others;
end behavioral;
```



All options must be covered and only one must be true!

- The "when others" clause can be used to ensure that all options are covered

Modeling Complex Behavior



Memory Module

add R1, R2, R3
sub R3, R4, R5
move R7, R3

•
•
•

Instruction Set Simulation

- Concurrent signal assignment statements can easily capture the gate level behavior of digital systems
- Higher level digital components have more complex behaviors
    - input/output behavior not easily captured by concurrent signal assignment statements
    - models utilize state information
    - incorporate data structures
- We need more powerful constructs

The Process Statement

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity mux4 is
port (   In0, In1, In2, In3  : in std_logic_vector  (7 downto 0);
         Sel: in std_logic_vector(1  downto  0);
         Z  : out std_logic_vector  (7 downto 0));
end mux4;

architecture behavioral-3 of mux4 is
begin
process (Sel, In0, In1, In2, In3)
variable Zout: std_logic;
begin
      if (Sel = “00”) then Zout:= In0;
      elsif (Sel = “01”) then Zout:= In1;
      elsif (Sel = “10”) then Zout:= In2;
      else Zout := In3;
      end if;
      Z <= Zout;
end process;
end behavioral;
```

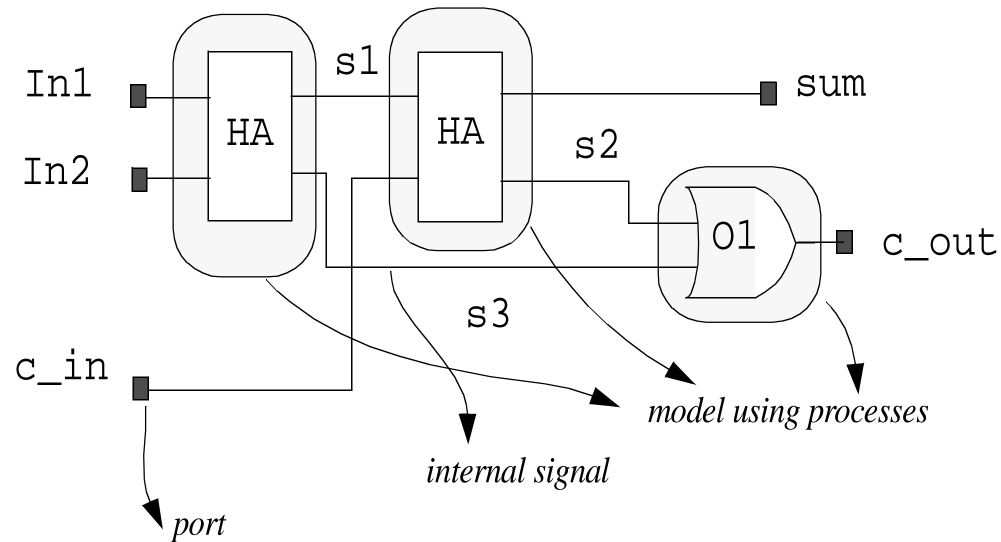
Sensitivity List

Variable Assignment

The Process Construct

- Statements in a process are executed sequentially
- A process body is structured much like conventional C or Pascal programs
  - declaration and use of variables
  - *if-then*, *if-then-else*, *case*, *loop* and *while* constructs
  - process can contain signal assignment statements
- A process executes concurrently with other concurrent signal assignment statements
- A process takes 0 simulation time to execute and may schedule events in the future: we can think of a process as a complex signal assignment statement!

Concurrent Processes: Full Adder



- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
- Processes communicate via signals

Concurrent Processes: Full Adder

```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port (In1, c_in, In2  : in std_logic;
       sum, c_out  : out std_logic);
end full_adder;

architecture behavioral of
full_adder is
signal  s1, s2, s3 : std_logic;
constant delay  :Time:= 5 ns;
begin
HA1: process (In1, In2)
begin
s1 <= (In1 xor In2) after delay;
s3  <= (In1  and In2) after delay;
end process HA1;

HA2: process(s1,c_in)
begin
sum <= (s1  xor c_in) after
delay;
s2 <= (s1 and c_in) after delay;
end process HA2;

OR1: process (s2, s3) -- process
describing the two-input OR gate
begin
c_out <= (s2 or s3) after delay;
end process OR1;
end behavioral;
```

Concurrent Processes: Half Adder

```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port (a, b : in std_logic;
sum, carry : out std_logic);
end half_adder;

architecture behavior  of
half_adder is
begin
sum_proc: process(a,b)
        begin
          if (a  =  b) then
               sum  <= '0' after 5 ns;
          else
               sum <= (a or b) after 5 ns;
          end if;
        end process;

 carry_proc: process (a,b)
begin
case a is
when '0' =>
carry  <= a  after  5  ns;
when '1' =>
carry  <=  b  after  5  ns;
when others =>
carry  <=  'X' after 5 ns;
end case;
end process carry_proc;
end behavior;
```

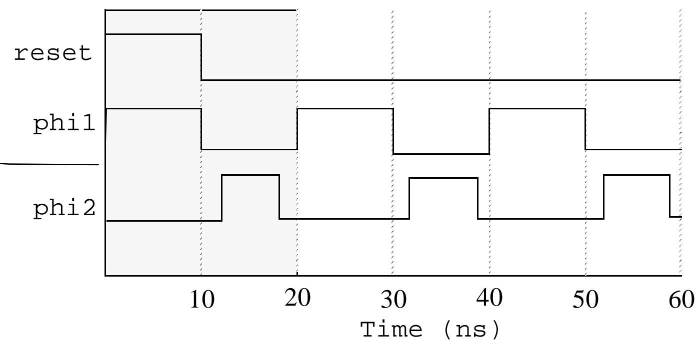
The Wait Statement

```
library IEEE;
use  IEEE.std_logic_1164.all;
entity dff is
port (D, Clk : in std_logic;
    Q, Qbar : out std_logic);
end dff;

architecture behavioral of dff is
begin
output: process
begin
wait until (Clk'event and Clk = '1'); -- wait for edge

   Q <= D after 5 ns;
   Qbar <= not D after 5 ns;
end process  output;
end behavioral;
```

- wait for <*time expression*>, wait on <*signal*>, wait until <*boolean expression*>

The Wait Statement: Waveform Generation

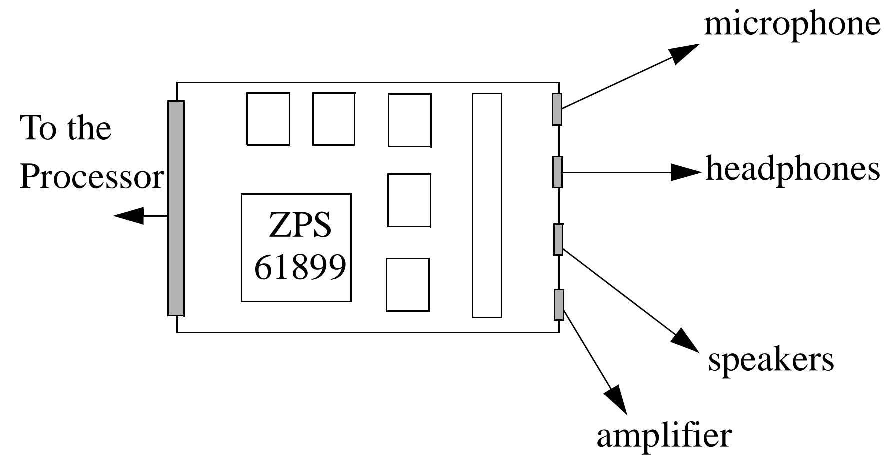
```
library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out
std_logic);
end two_phase;

architecture behavioral of
two_phase is
begin
rproc: reset <= '1', '0' after 10 ns;
clock_process: process
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns, '0' after
18 ns;
wait for 20 ns;
end process clock_process;
end behavioral;
```



The Wait Statement

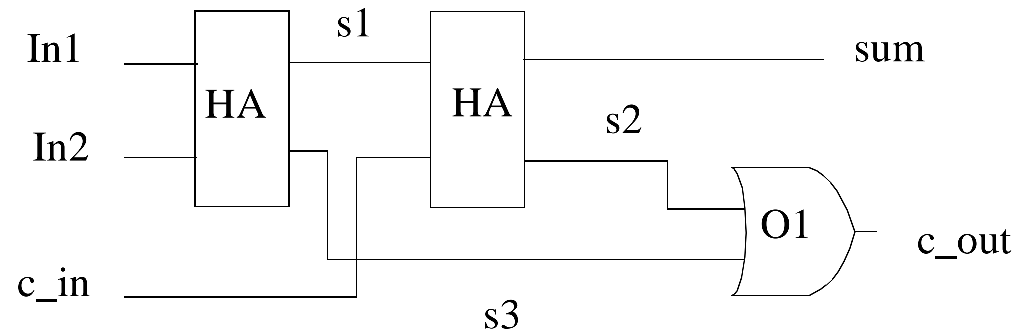
- A process can have multiple wait statements
- A process cannot have both a wait statement and a sensitivity list (it should have one or the other)

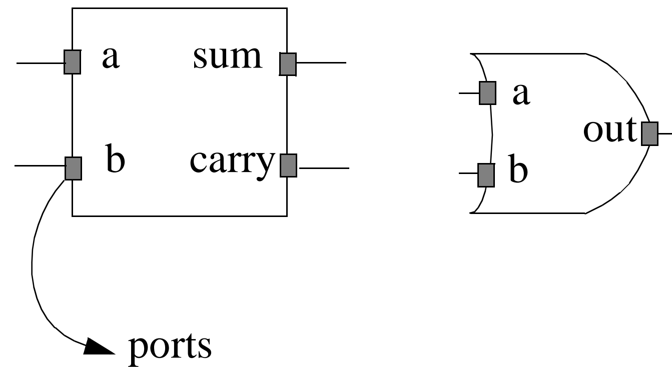
Modeling Structure



- Structural models describe a digital system as an interconnection of components
- Descriptions of the components must be available as structural or behavioral models

Modeling Structure





- Define the components used in the design
- Describe the interconnection of these components

Modeling Structure

```
architecture structural of full_adder is
component half_adder                    -- the declaration
port (a, b: in std_logic;               -- of components you will use
      sum, carry: out std_logic);
end component;

component  or_2
 port(a, b : in std_logic;
         c : out std_logic);
end component;

signal s1, s2, s3 : std_logic;
begin
H1: half_adder port map (a => In1, b => In2, sum=>s1, carry=>s3);
H2:half_adder port map (a => s1, b => c_in, sum =>sum,
                        carry => s2);
O1:  or_2 port map (a => s2, b => s3, c => c_out);
end structural;
```

unique name of the components

component type

interconnection of the component ports

component instantiation statement

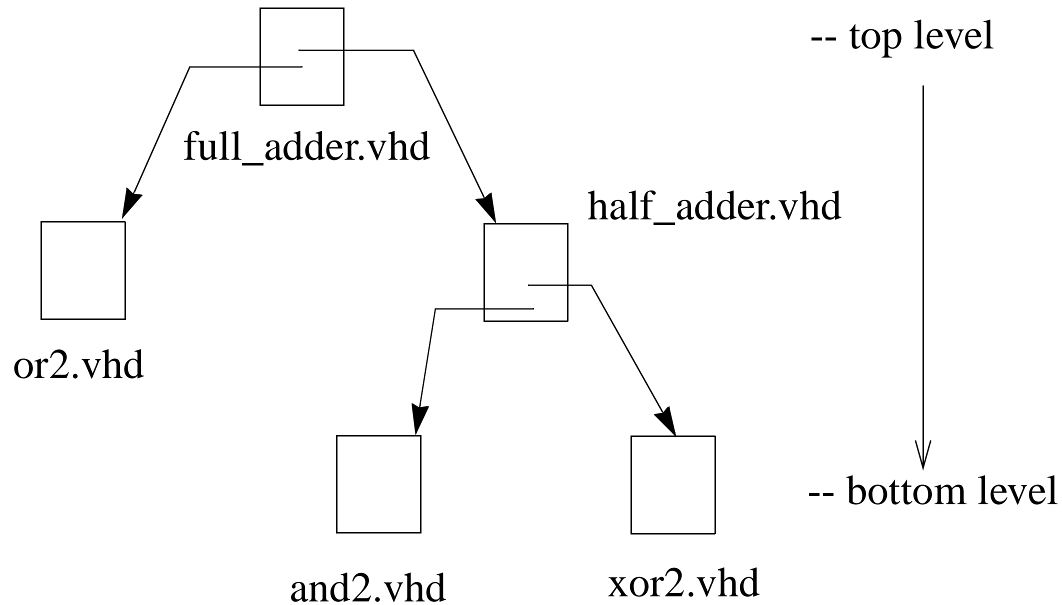
- Entity/architecture for half_adder and or_2 must exist

Hierarchy and Abstraction

```
architecture structural of half_adder is
component xor2
 port (a, b  : in std_logic;
        c : out std_logic);
end component;
component and2
port (a, b : in std_logic;
          c  : out std_logic);
end component;
begin
EX1: xor2 port map (a => a, b => b, c => sum);
AND1: and2 port map (a=> a, b=> b, c=> carry);
end structural;
```

- Structural descriptions can be nested
- The half adder may itself be a structural model

Hierarchy and Abstraction



- Nested structural descriptions to produce hierarchical models
- The hierarchy is flattened prior to simulation
- Behavioral models of components at the bottom level must exist

Generics

```
library IEEE;
use IEEE.std_logic_1164.all;

entity xor2 is
generic (gate_delay : Time:= 2 ns);
port(In1, In2 : in std_logic;
     z : out std_logic);
end xor2;

architecture behavioral of xor2 is
begin
z <= (In1 xor In2) after gate_delay;
end behavioral;
```

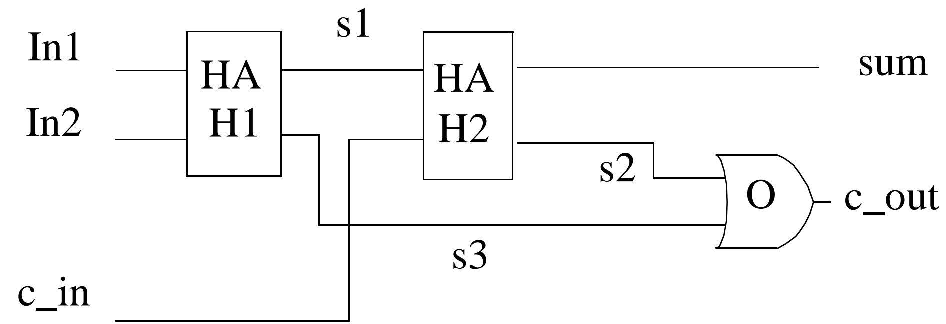
- Enables the construction of parameterized models

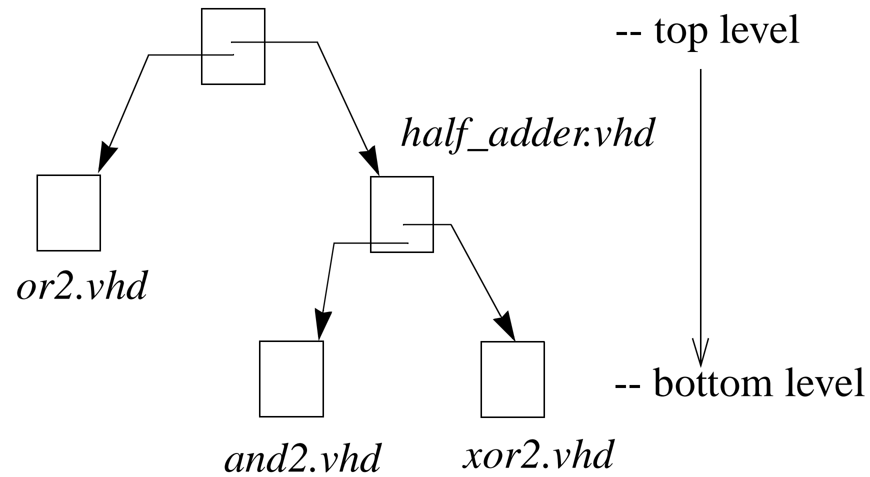
Generics in Hierarchical Models

```
architecture generic_delay of half_adder is
component xor2
generic (gate_delay: Time);
port (a, b  : in std_logic;
      c : out std_logic);
end component;

component and2
generic (gate_delay: Time);
port (a, b : in std_logic;
      c  : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => 6 ns)
           port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 3 ns)
          port map(a=> a, b=> b, c=> carry);
end generic_delay;
```

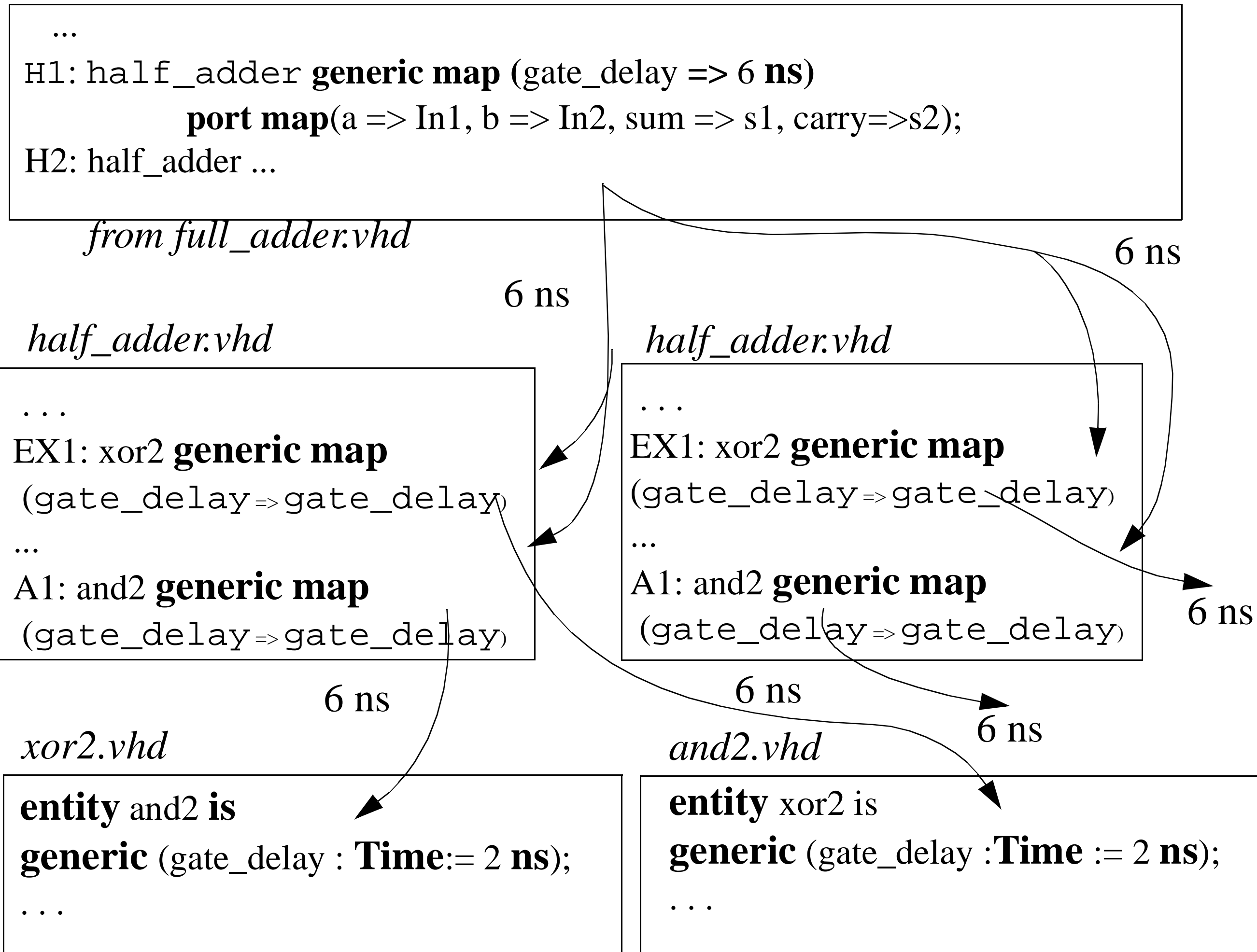
- Parameter values are passed through the hierarchy

Example: Full Adder





Example: Full Adder



Precedence of Generic Declarations

```
architecture generic_delay2 of half_adder is
component xor2
generic (gate_delay: Time);
 port(a,b : in std_logic;
       c : out std_logic);
end component;

component and2
generic (gate_delay: Time:= 6 ns);
port (a, b : in std_logic;
   c : out std_logic);
end component;

begin
EX1: xor2 generic map (gate_delay => gate_delay)
        port map(a => a, b => b, c => sum);
A1: and2 generic map (gate_delay => 4 ns)
        port map(a=> a, b=> b, c=> carry);
end generic_delay2;
```

takes precedence

- Generic map takes precedence over the component declaration

Further reading

- VHDL Starter's Guide, Sudhakar Yalamanchili
- Check the VHDL references link from the course web page
- Look at the on-line documentation available with ISE – there is a huge VHDL manual
- There are many on-line VHDL tutorials available

Homework

1. **Download and install WebPack and ModelSim as described on the VHDL references course web page,**

OR

Access the ISE Project Manager from a *vmware* enabled school Linux workstation;

AND

2. **Familiarize yourself with the design environment by completing the Tutorial and testing the designs we have discussed in lectures.**