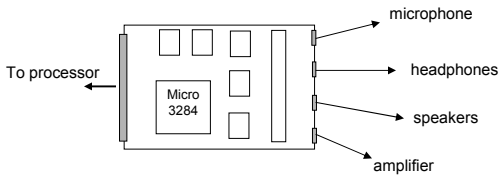
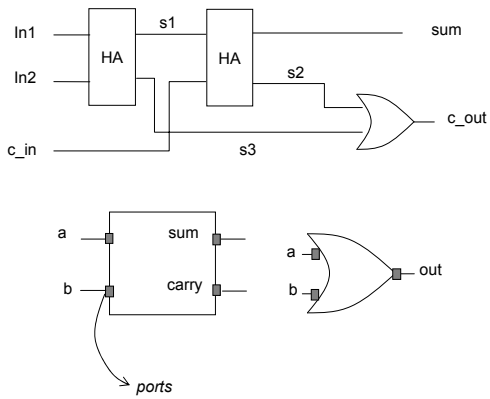
Modeling Structure

Elements of Structural Models



- Structural models describe a digital system as an interconnection of components
- Descriptions of the behavior of the components must be independently available as structural or behavioral models
  - An entity/architecture for each component must be available

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 is -- the declaration
port (a, b: in std_logic; -- of components you will use
sum, carry: out std_logic);
end component half_adder;
component or_2 is
port(a, b : in std_logic;
c : out std_logic);
end component or_2;
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 architecture 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

Example: State Machine

```
library IEEE;
use IEEE.std_logic_1164.all;
entity serial_adder is
port (x, y, clk, reset : in std_logic;
z : out std_logic);
end entity serial_adder;
architecture structural of serial_adder is
--
-- declare the components that we will be using
--
component comb is
port (x, y, c_in : in std_logic;
z, carry : out std_logic);
end component comb;
component dff is
port (clk, reset, d : in std_logic;
q, qbar : out std_logic);
end component dff;
signal s1, s2 :std_logic;
begin
--
-- describe the component interconnection
--
C1: comb port map (x => x, y => y, c_in =>
s1, z =>z, carry => s2);
D1: dff port map(clk => clk, reset =>reset,
d=> s2, q=>s1,
qbar => open);
end architecture structural;
```

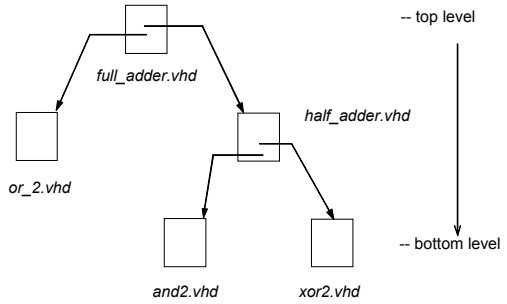
- Structural models can be easily generated from schematics
- Name conflicts in the association lists?
- The “open” attribute

Hierarchy and Abstraction

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

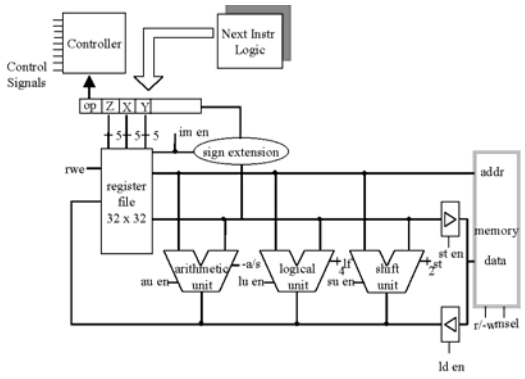
- Structural descriptions can be nested
- The half adder may itself be a structural model

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

Hierarchy and Abstraction



- Use of IP cores and vendor libraries
- Simulations can be at varying levels of abstraction for individual components

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 entity xor2;

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

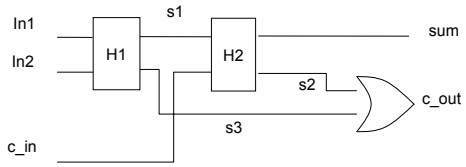
- Enables the construction of parameterized models

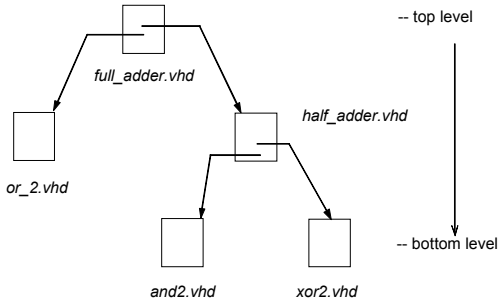
Generics in Hierarchical Models

```
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 architecture generic_delay;
```

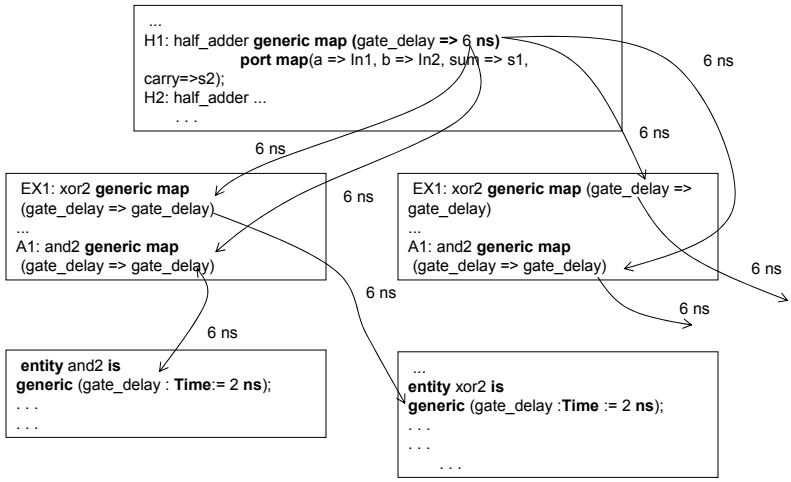
- Parameter values are passed through the hierarchy

Example: Full Adder





Example: Full Adder



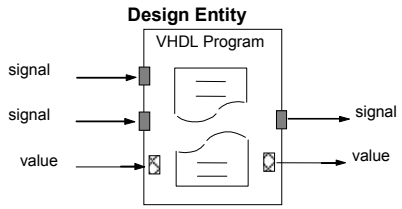
Precedence of Generic Declarations

```
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

Generics: Properties



- Generics are constant objects and can only be read
- The values of generics must be known at compile time
- They are a part of the interface specification but do not have a physical interpretation
- Use of generics is not limited to “delay like” parameters and are in fact a very powerful structuring mechanism

Example: N-Input Gate

```
entity generic_or is
generic (n: positive:=2);
port (in1 : in std_logic_vector ((n-1) downto 0);
z : out std_logic);
end entity generic_or;
architecture behavioral of generic_or is
begin
process (in1) is
variable sum : std_logic:= '0';
begin
sum := '0'; -- on an input signal transition sum must
be reset
for i in 0 to (n-1) loop
sum := sum or in1(i);
end loop;
z <= sum;
end process;
end architecture behavioral;
```

- Map the generics to create different size OR gates

Example: Using the Generic N-Input OR Gate

```
architecture structural of full_adder is
component generic_or
generic (n: positive);
port (in1 : in std_logic_vector ((n-1) downto 0);
z : out std_logic);
end component;
...
... -- remainder of the declarative region from earlier example
...
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: generic_or      generic map (n => 2)
                    port map (a => s2, b => s3, c => c_out);
end structural;
```

- Full adder model can be modified to use the generic OR gate model via the *generic map ()* construct
- Analogy with macros

Example: N-bit Register

```
entity generic_reg is
generic (n: positive:=2);
port (      clk, reset, enable : in std_logic;
            d : in std_logic_vector (n-1 downto 0);
            q : out std_logic_vector (n-1 downto 0));
end entity generic_reg;
architecture behavioral of generic_reg is
begin
reg_process: process (clk, reset)
begin
 if reset = '1' then
            q <= (others => '0');
 elsif (rising_edge(clk)) then
            if enable = '1' then q <= d;
            end if;
 end if;
end process reg_process;
end architecture behavioral;
```

- This model is used in the same manner as the generic OR gate

Component Instantiation and Synthesis

- Design methodology for inclusion of highly optimized components or "cores"
  - Optimized in the sense of placed and routed
  - Intellectual property cores for sale
  - Check out http://www.xilinx.com/ipcenter/index.htm
- Core generators for static generation of cores
  - Generation of VHDL/Verilog models of placed and routed designs
  - Component instantiation for using cores in a design
- Access to special components/circuitry within the target chip

Example: Using Global Resources

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

entity my_clocks is
port (phi1, phi2: out std_logic);
end entity my_clocks;

architecture behavioral of my_clocks is
component OSC4 is -- on chip oscillator
port (F8M : out std_logic; -- 8 Mhz clock
F500k : out std_logic;-- 500Khz clock
F15 : out std_logic);-- 15 hz clock
end component OSC4;

component BUFG is -- global buffer connection
    to low skew lines
port (I : in std_logic;
      O : out std_logic);
end component BUFG;

signal local_F15, local_phi2 : std_logic; -- local
    signals
begin
O1: osc4 port map(F15 =>local_F15); --
    instantiate the oscillator
B1: bufg port map (I => local_F15, O => phi1); --
    instantiate the two global buffers
B2: bufg port map (I => local_phi2, O => phi2);

local_phi2 <= not local_F15; -- phi2 is the
    complement of phi1

end architecture behavioral;
```

- Component abstractions for special support within the chip
  - For global signals
  - For shared system level components

Implementing Global Set/Reset in VHDL

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

entity sig_var is
port ( clk, reset,x, y, z: in std_logic;
w : out std_logic);
end sig_var;

architecture behavior of sig_var is
component STARTUP is
port (GSR : in std_logic);
end component STARTUP;
signal s1, s2 : std_logic;
begin
U1: STARTUP port map(GSR => reset);

process
begin
wait until (rising_edge(clk));
if (reset = '1') then
w <= '0';
s1 <= '0';
s2 <= '0';
else
L1: s1 <= x xor y;
L2: s2 <= s1 or z;
L3: w <= s1 nor s2;
end if;
 end process;
end behavior;
```

- GSR inferencing optimization
- Connect your reset signal to this global net
- Note
  - improves “routability” of designs

Core Generators

- Xilinx Logic Core utility
- Parameterized modules
  - User controlled generation of VHDL modules
  - Instantiation within a design
  - Simulaton and synthesis
- Third party view of the world of hardware design
  - Analogy with software and compilers
  - What is software vs. hardware anymore?

The Generate Statement

- What if we need to instantiate a large number of components in a regular pattern?
  - Need conciseness of description
  - Iteration construct for instantiating components!
- The *generate* statement
  - A parameterized approach to describing the regular interconnection of components

```
a: for i in 1 to 6 generate
        a1: one_bit  generic map (gate_delay)
        port map(in1=>in1(i), in2=> in2(i), cin=>carry_vector(i-1),
         result=>result(i), cout=>carry_vector(i),opcode=>opcode);
end generate;
```

The Generate Statement: Example

- Instantiating an register

```
entity dregister is
port ( d : in std_logic_vector(7 downto 0);
q : out std_logic_vector(7 downto 0);
clk : in std_logic);
end entity dregisters
architecture behavioral of dregister is
begin
d: for i in dreg'range generate
reg: dff port map( (d=>d(i), q=>q(i), clk=>clk;
end generate;
end architecture register;
```

- Instantiating interconnected components
  - Declare local signals used for the interconnect

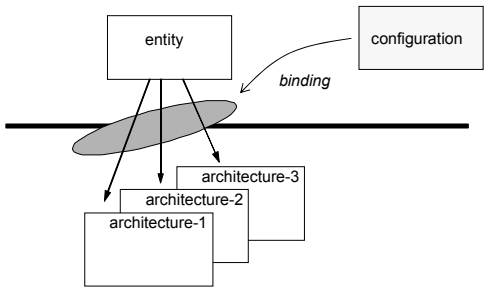
The Generate Statement: Example

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

entity multi_bit_generate is
generic(gate_delay:time:= 1 ns;
          width:natural:=8); -- the default is a 8-bit ALU
port( in1 : in std_logic_vector(width-1 downto 0);
       in2 : in std_logic_vector(width-1 downto 0);
       result : out std_logic_vector(width-1 downto 0);
       opcode : in std_logic_vector(1 downto 0);
       cin : in std_logic;
       cout : out std_logic);
end entity multi_bit_generate;

architecture behavioral of multi_bit_generate is

component one_bit is   -- declare the single bit ALU
generic (gate_delay:time);
port (in1, in2, cin : in std_logic;
      result, cout : out std_logic;
      opcode: in std_logic_vector (1 downto 0));
end component one_bit;

signal carry_vector: std_logic_vector(width-2 downto 0);
-- the set of signals for the ripple carry

begin
a0:  one_bit generic map (gate_delay) -- instantiate ALU
for bit position 0
port map (in1=>in1(0), in2=>in2(0), result=>result(0),
cin=>cin, opcode=>opcode, cout=>carry_vector(0));

a2to6: for i in 1 to width-2 generate  -- generate
instantiations for bit positions 2-6
a1: one_bit  generic map (gate_delay)
port map(in1=>in1(i), in2=> in2(i), cin=>carry_vector(i-1),
result=>result(i), cout=>carry_vector(i),opcode=>opcode);
end generate;

a7: one_bit generic map (gate_delay) -- instantiate ALU
for bit position 7
port map (in1=>in1(width-1), in2=>in2(width-1), result=>
result(width-1), cin=>carry_vector(width-2),
opcode=>opcode, cout=>cout);
end architecture behavioral;
```

Using the Generate Statement

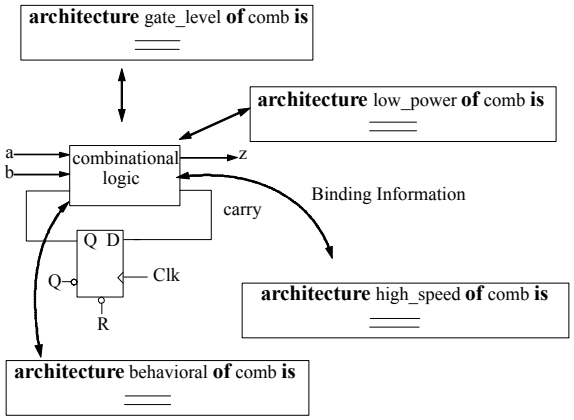
- Identify components with regular interconnect
- Declare local arrays of signals for the regular interconnections
- Write the generate statement
  - Analogy with loops and multidimensional arrays
  - Beware of unconnected signals!
- Instantiate remaining components of the design

Configurations



- A design entity can have multiple alternative architectures
- A configuration specifies the architecture that is to be used to implement a design entity

Component Binding



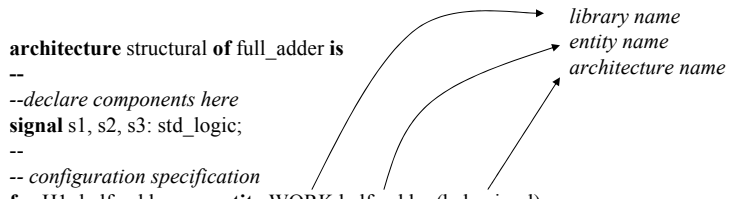
- We are concerned with configuring the architecture and not the entity
- Enhances sharing of designs: simply change the configuration

Default Binding Rules

```
architecture structural of serial_adder is
component comb is
port (a, b, c_in : in std_logic;
z, carry : out std_logic);
end component comb;

component dff is
port (clk, reset, d :in std_logic;
q, qbar :out std_logic);
end component dff;
signal s1, s2 : std_logic;

begin
C1: comb port map (a => a, b => b, c_in => s1, z =>z, carry => s2);
D1: dff port map(clk => clk, reset =>reset, d=> s2, q=>s1, qbar =>open);
end architecture structural;
```

- Search for entity with the same component name
- If multiple such entities exist, bind the last compiled architecture for that entity
- How do we get more control over binding?

Configuration Specification




```
architecture structural of full_adder is
--
--declare components here
signal s1, s2, s3: std_logic;
--
-- configuration specification
for H1: half_adder use entity WORK.half_adder (behavioral);
for H2: half_adder use entity WORK.half_adder (structural);
for O1: or_2 use entity POWER.lpo2 (behavioral)
generic map(gate_delay => gate_delay)
port map (I1 => a, I2 => b, Z=>c);
begin -- component instantiation statements
H1: half_adder port map (a =>In1, b => In2, sum => s1, carry=> s2);
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;
```

- We can *specify* any binding where ports and arguments match

Configuration Specification

- Short form where applicable

  for all: half_adder **use entity** WORK.half_adder (behavioral);

- Not constrained by the name space
- Delayed binding when a specification is not present
  - Will be available at a later step
  - Analogous to unresolved symbol references during compilation of traditional programs

Configuration Declaration

```
configuration Config_A of full_adder is -- name the configuration
-- for the entity
for structural -- name of the architecture being configured
for H1: half_adder use entity WORK.half_adder (behavioral);
end for;
--
for H2: half_adder use entity WORK.half_adder (structural);
end for;
--
for O1: or_2 use entity POWER.lpo2 (behavioral)
generic map(gate_delay => gate_delay)
port map (I1 => a, I2 => b, Z=>c);
end for;
--
end for;
end Config_A;
```

- Written as a separate design unit
- Can be written to span a design hierarchy
- Use of the “for all” clause

Summary

- Structural models
  - Syntactic description of a schematic
- Hierarchy and abstraction
  - Use of IP cores
  - Mixing varying levels of detail across components
- Generics
  - Construct parameterized models
  - Use in configuring the hardware
- Configurations
  - Configuration specification
  - Configuration declaration