Structural Modeling

- Behavioral Modeling (describing internal behavior of a component)
  - CSA statements (dataflow model)
  - Process
- Structural Modeling
  - Describes how the components are connected.
  - Behavioral models are assumed to exist in local working directory or in a library.
  - Structural modeling facilitate the use of hierarchy and abstraction in modeling complex systems
  - Structural models can be integrated into models that also use processes and CSA statements
Describing Structure

- **Block diagrams**
  - Components are represented by blocks interconnected by lines representing signals.

- **Full Adder**
Full Adder

- We assume that we have the description of half adders and or gate.
  - Actually, we are not interested in the internal behavior of these components.
- To connect them, what we need is the interface of each component.

Each component, signal and ports of components have names
Connecting Components

- List of components
- Uniquely labeling components to distinguish among multiple instances of the same component.
- Set of signals connecting them
- Specify how signals are connected to ports of components.
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 entity full_adder;

architecture structural of full_adder is
component half_adder is
port (x, y: in std_logic;
      sum, carry: out std_logic);
end component half_adder;

component or_2 is
port (x, y: in std_logic;
      z: out std_logic);
end component or_2;

signal s1, s2, s3: std_logic;
begin
  H1: half_adder port map (x=>in1, y=>in2, sum=>s1, carry=>s3);
  H2: half_adder port map (x=>s1, y=>c_in, sum=>sum, carry=>s2);
  O1: or_2 port map (x=>s2, y=>s3, z=>c_out);
end architecture structural;
**Structural Model**

- Different instances of a component have identical input port names.
  - This is not a problem; since the port name is associated with a specific instance through a unique label.

- Components that are going to be used are declared within the declarative region of the architecture
  - The actual description must be placed somewhere that CAD tools can locate: e.g. working directory, a library, or in some directory whose path is in the search path of CAD tools.
Structural Model of a State Machine

- Bit-serial adder
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

component comb is
port (x,y, c_in: in std_logic;
     z, carry: out std_logic);
end component comb;

component dff is
port (D,clk, R: in std_logic;
     Q, Qbar: out std_logic);
end component dff;

signal s1, s2: std_logic;
begin
C1: comb port map(x=>x, y=>y, c_in=>s1, z=>z, carry=>s2);
D1: dff port map(D=>s2, clk=>clk, R=>reset, Q=>s1, Qbar=>open);
end architecture structural;

We cannot assume that port names are unique.
Constructing Structural VHDL Models

• Two Steps:
  - Drawing the annotated schematics
  - Converting it to VHDL

• Make sure that description of components you will use is available.
  - i.e. a working entity-architecture pair.
  - Using entity descriptions, create a block for each component.

• Connect each port of every component to the port of another entity or to the input or output port of the entity being modeled.

• Label each component
• Label each internal signal
• Label each system input and output port, define its mode and type
library library-name-1, library-name-2;
use library-name-1.package-name.all;
use library-name-2.package-name.all;

entity entity_name is
  port(input signals: in type;
       output signals: out type);
end entity entity_name;

architecture structural of entity_name is
  -- declare components used
component component1-name is
  port (input signals: in type;
        output signals: out type);
end component component1-name;

component component2-name is
  port (input signals: in type;
        output signals: out type);
end component component2-name;
  -- declare all signals used to connect the components.
signal internal signals: type:=initialization;
begin
  -- label each component and connect its ports to signals or entity ports
  Label1: component1-name port map(port=>signal, ...);
  Label2: component2-name port map(port=>signal, ...);
end architecture structural;
Adder Design with Structural VHDL

1st Clock Cycle

(1111) + (0001)

worse case addition

2nd Clock Cycle

3rd Clock Cycle

4th Clock Cycle

Carry-ripple adder
A Faster Adder

- To add two corresponding bits of A and B, we have to know the carry from the sum of previous bits.
  - $c_1 = b_0c_0 + a_0c_0 + a_0b_0$
  - $c_2 = b_1c_1 + a_1c_1 + a_1b_1 = (b_1+a_1)(b_0c_0+a_0c_0+a_0b_0)+a_1b_1$
  - $c_3 = b_2c_2 + a_2c_2 + a_2b_2 = (b_2+a_2)(b_1+a_1)(b_0c_0+a_0c_0+a_0b_0)+a_1b_1+a_2b_2$
  - $c_4 = b_3c_3 + a_3c_3 + a_3b_3 = (b_3+a_3)(b_2+a_2)(b_1+a_1)(b_0c_0+a_0c_0+a_0b_0) + a_1b_1 + a_2b_2 + a_3b_3$

- Can you imagine the length of the expression for $c_{32}$?
Carry Lookahead - 1

• An approach in-between the two extremes

• Motivation:
  - If we didn’t know the value of carry-in, what could we do?
  - When would a carry be always generated?
    \[ g_i = a_i b_i \]
  - When would the carry be propagated?
    \[ p_i = a_i + b_i \]

• Using them we can write the following formula
  \[ c_{i+1} = g_i + p_i c_i \]

• To see where the signals get their names suppose
  \[ g_i = 1 \quad \Rightarrow \quad c_{i+1} = 1 + p_i c_i = 1 \]
  that is, the adder generates the CarryOut (independent of CarryIn)
• $g_i = 0$ and $p_i = 1 \Rightarrow c_{i+1} = c_i$

• That is CarryIn is propagated to CarryOut.

\[
c_1 = g_0 + p_0 c_0 \\
c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0 \\
c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\]

• **Question**: How can we compute $p_i$s and $g_i$s efficiently?
Brent-Kung Formulation

- An associative operation:
  \((x, y) \cdot (w, z) = (x+yw, yz)\)
- Apply this operation to the \(p_i\)s and \(g_i\)s

  - \((c_0, 1)\)
  - \((g_0, p_0) \cdot (c_0, 1) = (g_0 + p_0c_0, p_0) = (c_1, p_0)\)
  - \((g_1, p_1) \cdot (c_1, p_0) = (g_1 + p_1c_1, p_1p_0) = (c_2, p_1p_0)\)
  - \((g_2, p_2) \cdot (c_2, p_1) = (g_2 + p_2c_2, p_2p_1p_0) = (c_2, p_2p_1p_0)\)
  - \(\ldots\)
  - \((g_i, p_i) \cdot (c_i, p_{i\ldots p_1p_0}) = \)
    \((g_i + p_ic_i, p_{i\ldots p_1p_0}) = (c_{i+1}, p_{i\ldots p_1p_0})\)
Brent-Kung Circuit

\[(g_i, p_i) \cdot (c_i, p_i \ldots p_1 p_0) = (g_i + p_i c_i, p_i \ldots p_1 p_0) = (c_{i+1}, p_i \ldots p_1 p_0)\]
Brent-Kung Formulation

\((g_i, p_i)\) \rightarrow \text{(c\_i, p\_i \ldots p\_0)}\)

\((g_1, p_1)\) \rightarrow \text{(c\_1, p\_0)}\)

\((g_0, p_0)\) \rightarrow \text{(c\_0, 1)}\)

Computing carries becomes a prefix problem
Prefix Problem

- **Input**: $x_0, x_1, x_2, \ldots, x_7$
- **Output**: $y_0, y_1, y_2, \ldots, y_7$
- $y_0 = x_0$
- $y_1 = x_1x_0$
- $y_2 = x_2x_1x_0$
- $y_3 = x_3x_2x_1x_0$
- $\ldots$
- $y_7 = x_7x_6\ldots x_1x_0$

- Various algorithms for prefix problem with different complexities.
Prefix Algorithms

- Sequential or Serial Algorithm

Depth: n-1
Size: n-1
Prefix Algorithms

- Obvious Parallelization Algorithm

Depth: ⌈log₂n⌉
Size: 0 + 1 + 2 + 3 + ... + n-1 = n(n-1)/2 = O(n²)
Ladner-Fischer Constructions

- **Depth:** $O(\log_2 n)$
- **Size:** $O(n)$
- Assumption: $n = 2^d$.
- $(n$ is the number of bits in our adder$)$
- Ladner-Fischer constructions provide a family of circuits for $n$:
  - $P_k(n)$ where $0 \leq k \leq d$ and $d = \log_2 n$.
- For example:
  - $n = 2 \ (d = 1) \Rightarrow P_0(2)$ and $P_1(2)$
  - $n = 4 \ (d = 2) \Rightarrow P_0(4), P_1(4),$ and $P_2(4)$.
  - $n = 8 \ (d = 3) \Rightarrow P_0(8), P_1(8), P_2(8),$ and $P_3(8)$
Ladner-Fischer Constructions

- Properties of $P_k(n)$ circuit:
  - Depth: $D_k(n) = d + k$
  - Size: $S_k(n) = 2(1+2^{-k})n - F(5+d-k) + 1 - k$
  where $F$ is the Fibonacci number

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(i)$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
</tr>
</tbody>
</table>

$S_0(n) = 3n - F(5+d) + 1$
$S_1(n) = 3n - F(4+d)$
Ladner-Fischer Circuits

- \( P_k(1) \) and \( k = 0 \Rightarrow P_0(1) \)
  \[ D = 0, S = 0 \]

- \( P_k(2) \) and \( k = 0, 1 \Rightarrow P_0(2) \) and \( P_1(2) \)
- \( P_0(2) \Rightarrow D = 1 \) and \( S = 1 \)
- \( P_1(2) \) does not exist since there is only one circuit to add two numbers.
- Thus \( P_1(2) = P_0(2) \)

\[
\begin{align*}
P_0(2) & \quad D = 1, S = 1
\end{align*}
\]
Ladner-Fischer Circuits

- $P_k(4)$ and $k = 0, 1, 2 \Rightarrow P_0(4)$, $P_1(4)$, and $P_2(4)$
- There is no other circuit but $P_0(4)$ for $n = 4$.
- Question: how to construct $P_0(4)$ from $P_0(2)$?

In our case these corresponds to Brent-Kung circuits

\[ P_0(2) \quad P_1(2) \]

\[ x_3 x_2 \quad x_1 x_0 \]

\[ x_3 x_2 x_1 x_0 \quad x_2 x_1 x_0 \quad x_1 x_0 \quad x_0 \]
$P_0(4)$ Circuit for Carry Calculation

- Brent-Kung Circuit
General Construction Scheme for $P_0(n)$

\[ \begin{align*}
&n-1 \quad \cdots \quad n/2 \\
&\quad \downarrow \\
&P_0(n/2) \\
&\quad \uparrow \\
&\quad \downarrow \\
&\quad \cdots \\
&1 \quad \cdots \quad 0 \\
&\quad \uparrow \\
&P_1(n/2) \\
&\quad \downarrow \\
&\quad \cdots \\
&\quad \downarrow \\
&\quad \cdots \\
&\quad \cdots
\end{align*} \]
General Construction Scheme for $P_{\perp}(n)$
\[ P_0(4) \]

\[ P_1(4) = P_0(4) \]

- \( 0 \rightarrow (c_0, 1) \)
- \( 1 \rightarrow (g_0, p_0) \)
- \( 2 \rightarrow (g_1, p_1) \)
- \( 3 \rightarrow (g_2, p_2) \)
- \( \ldots \)
- \( 7 \rightarrow (g_6, p_6) \)

- \( 10 \rightarrow (c_1, p_0) \)
- \( 210 \rightarrow (c_2, p_1p_0) \)
- \( 3210 \rightarrow (c_3, p_2p_1p_0) \)
- \( \ldots \)
- \( 7 \rightarrow 210 \rightarrow (c_7, p_6 \ldots p_1p_0) \)
Assignment

• Construct CLA adders of 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit using structural VHDL modeling.
  1. create a behavioral model of Brent-Kung circuit.
  2. construct $P_0(2)$
  3. construct $P_0(4)$ using $P_0(2)$
  4. construct $P_0(8)$ using $P_0(4)$
  5. construct $P_1(8)$ using $P_0(4)$
  6. construct $P_0(16)$ using $P_0(8)$ and $P_1(8)$
  7. construct $P_1(16)$ using $P_0(8)$
  8. construct $P_0(32)$ using $P_0(16)$ and $P_1(16)$
  9. construct $P_1(32)$ using $P_0(16)$
  10. construct $P_0(64)$ using $P_0(32)$ and $P_1(32)$
Hierarchy of models used in the full adder
At lowest level, a behavioral description must exist
Structural Model of a Half Adder

architecture structural of half_adder is

component xor2 is
port (x, y: in std_logic;
     z: out std_logic);
end component xor2;

component and2 is
port (x, y: in std_logic;
     z: out std_logic);
end component and2;

begin
EX1: xor2 port map(x=>x, y=>y, z=>sum);
AND1: and2 port map(x=>x, y=>y, z=>carry)
end architecture structural;

• There may be a many levels of hierarchy in structural model.
• No behavioral description.
Flattening

• When the simulator loads the hierarchy of structural model, it replace each component by its behavioral description.
  - This process is repeated at each level hierarchy.
• This is called as flattening of hierarchy.
• Flattening of the all hierarchy may result in a very complicated behavioral description of a very complex design.
  - Simulation may take too long
  - Preserving hierarchy in a structural model is a way of managing very complex systems.
  - Components can be simulated in isolation.
Parameterized Models

• **Aim**: to construct parameterized models.
  - You use 5 ns delay for your 2-input and gates in your model.
  - Thanks to developments in the technology, you would like to simulate your model using 1 ns propagation delay for your 2-input and gates.
  - It is hard to change the delay of every 2-input gate delay manually.
  - **Solution**: to be able to decide delay values at simulation time rather than design time.

• Parameterized models are useful in creating libraries of models that can be shared and reused.
library IEEE;
use IEEE.std_logic_1164.all

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

architecture behavioral of xor2 is
begin
  z <= (x and y) after gate_delay;
end architecture behavioral;

- The default delay for xor2 gate is 2 ns.
- The gate delay can be specified at the time of its use.
architecture structural of half_adder is

component xor2 is
generic (gate_delay: Time);
port (x,y: in std_logic;
     z: out std_logic);
end component xor2;

component and2 is
generic (gate_delay: Time:=3 ns);
port (x, y: in std_logic;
     z: out std_logic);
end component and2;

begin
EX1: xor2 generic map (gate_delay => 6 ns)
     port map (x=>x, y=>y, z=>sum);
AND1: and2 port map (x=>x, y=>y, z=>carry)
end architecture structural;
Specifying Generics Constants

1. in the component declaration
   
   ```
   component and2 is
       generic (gate_delay: Time := 3 ns);
       port (x, y: in std_logic; z: out std_logic);
   end component and2;
   ```

2. in the component instantiation
   
   ```
   EX1: xor2 generic map (gate_delay => 6 ns)
       port map (x=>x, y=>y, z=>sum);
   ```

   - If both specified, the value provided in the component instantiation takes precedence.
   - If none is specified, the default value defined in the entity is used.
Passing Generic Values through Hierarchy

library IEEE;
use IEEE.std_logic_1164.all

entity half_adder is
generic(gate_delay: Time:= 3ns);
port(a, b: in std_logic; sum, carry: out std_logic);
end entity half_adder;

architecture structural of half_adder is

component xor2 is
generic(gate_delay: Time);
port(x, y: in std_logic; z: out std_logic);
end component xor2;

component and2 is
generic(gate_delay: Time);
port(x, y: in std_logic; z: out std_logic);
end component and2;

begin
EX1: xor2 generic map(gate_delay=>gate_delay)
    port map(x=>a, y=>b, z=>sum);
A1: and2 generic map(gate_delay=>gate_delay)
    port map(x=>a, y=>b, z=>carry);

end architecture structural;
Passing Generic Values through Hierarchy

The generic value coming from the upper level of hierarchy overrides the other values.
N-input AND Gate

entity and_gate is
generic(n:natural:=2);
port(x: in std_logic_vector(n-1 downto 0);
    z: out std_logic);
end entity and_gate;

architecture generic of and_gate is
begin
process(x)
variable and_out:=‘1’;
begin
and_out <= ‘1’; --on an input signal transition it must be set to 1
for i in 1 to n loop
    and_out := and_out and x(i);
    exit when and_out = ‘0’;
end loop;
z <= and_out;
end process;
end architecture generic;

entity or_gate is
generic(m:integer:=2);
port(x: in std_logic_vector(m downto 1);
    z: out std_logic);
end entity or_gate;
entity another_gen_ex is
end entity another_gen_ex;

architecture generic of another_gen_ex is
-- component declaration for the gates
component and_gate is
generic (n: natural := 5);
port (x: in std_logic_vector(n-1 downto 0); z: out std_logic);
end component;

component or_gate is
generic (m: integer);
port (x: in std_logic_vector(m downto 1); z: out std_logic);
end component;
signal s1, s2, s3, s4: std_logic;
signal sa: std_logic_vector(1 to 5);
signal sb: std_logic_vector(2 downto 1);
signal sc: std_logic_vector(1 to 10);
signal sd: std_logic_vector(5 downto 0);
begins
-- component instantiations
o1: or_gate generic map(6) port map(sd, s1);
a1: and_gate generic map(N => 10) port map(sc, s3);
a2: and_gate port map(sa, s4);
-- o2: or_gate port map(sb, s2); -- illegal
end architecture generic;
N-bit Register

library IEEE;
use IEEE.std_logic_1164.all;

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) is
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;

Note the statement q <= (others => '0');
Trace of the Operation of a 2-bit Register
Component Instantiation & Synthesis

- Structural design
  - The ability to instantiate specific components enables us to use previously synthesized and optimized implementation of common components
  - Those implementation may be provided by the vendors, other design groups, or by yourself from a previous design.
  - Performance critical components may be manually placed and routed to a specific target device.
  - Such components may be provided by vendors in design libraries
Core Generators

- **A CAD tool**
  - allow you select and then configure the components of your choice.
  - The CAD tool will then generate a synthesizeable model of this component and corresponding VHDL model.
  - The CAD tools generally do a good job of producing models that will generate highly optimized implementations for the target FPGA device.
  - The LogiBLOX of Xilinx Foundation is such a tool.
Xilinx Foundation LogiBLOX
addsub8.vhi File

-- LogiBLOX ADD_SUB Module "addsub8"
-- Created by LogiBLOX version E.35
-- on Sun Nov 02 17:37:55 2003
-- Attributes
-- MODTYPE = ADD_SUB
-- BUS_WIDTH = 8
-- OPTYPE = ADD_SUB
-- REGISTERED = NONE
-- ENCODING = SIGNED
-- STYLE = MAX_SPEED

------------------------------------------------------------------------
-- Component Declaration
--------------------------------------------------------------------------
component addsub8
  PORT(
    ADD_SUB: IN std_logic;
    C_IN: IN std_logic;
    A: IN std_logic_vector(7 DOWNTO 0);
    B: IN std_logic_vector(7 DOWNTO 0);
    SUM: OUT std_logic_vector(7 DOWNTO 0);
    OVFL: OUT std_logic;
    C_OUT: OUT std_logic);
  end component;
--------------------------------------------------------------------------

-- Component Instantiation
--------------------------------------------------------------------------
instance_name : addsub8 port map
  (ADD_SUB => , C_IN => , A => , B => , SUM => , OVFL => , C_OUT => );
• **Caveat:** Since we do not have the license, LogiBLOX will only provide the model in EDF format.

• In the previous example, EDF format is saved in addsub8.EDO file.