DIGITAL LOGIC DESIGN
VHDL Coding for FPGAs
Unit 4

✓ STRUCTURAL DESCRIPTION

- Hierarchical design: port-map, for-generate, if-generate.
- Examples: Adder, multiplier, ALU, Look-up Table.
- Introduction to Parametric Coding.
STRUCTURAL DESCRIPTION

- It is the generalization of the Concurrent Description. The circuits are described via interconnection of its subcircuits. This subcircuits can be described in concurrent code and/or sequential code.

- **Example:** Multiplexor 2-to-1.

- This case is trivial, since the interconnection is realised via logic operators, but nevertheless it is an example of structural description.
**STRUCTURAL DESCRIPTION**

- **Example:** 4-to-16 decoder

- We can describe this decoder in an structured way based on 2-to-4 decoders.

- However, we can also describe the 4-to-16 decoder using the with-select statement.
**STRUCTURAL DESCRIPTION**

- **Example:** DLX Processor

- In this type of systems, it is best to describe each component first, then assemble them to make the large system.

- We do not need to see such large system to realise the importance of the Structural Description.
**STRUCTURAL DESCRIPTION**

- Many systems can be described entirely in one single block: we can use the behavioral description, and/or concurrent statements (with-select, when-else).
- However, it is advisable not to abuse of this technique since it makes: i) the code less readable, ii) the circuit verification process more cumbersome, and iii) circuits improvements less evident.
- The structural description allows for a hierarchical design: we can ‘see’ the entire circuit as the pieces it is made of, then identify critical points and/or propose improvements on each piece.
- It is always convenient to have basic building blocks from which we can build more complex circuits. This also allows building block (or sub-system) to be re-used in a different circuit.
**STRUCTURAL DESCRIPTION**

- **Example:** 4-bit add/sub for numbers in 2’s complement.

- The circuit can be described in one single block. However, it is best to describe the Full Adder as a block in a separate file (*full_add.vhd*), then use as many full adders to build the 4-bit adder.

- The place where we use and connect as many full adders as desired, and possibly add extra circuitry is called the ‘top file’ (*my_addsub.vhd*). This creates a hierarchy of files in the VHDL project:
4-bit 2’s complement Adder

- Full Adder: VHDL Description (`fulladd.vhd`):

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity fulladd is
  port ( cin, x, y: in std_logic;
         s, cout: out std_logic);
end fulladd;

architecture struct of fulladd is
begin
  s <= x xor y xor cin;
  cout <= (x and y) or (x and cin) or (y and cin);
end struct;
```

---

Daniel Llamocca
4-bit 2’s complement Adder

- Top file (my_addsub.vhd): We need 4 full adders block and extra logic circuitry.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity my_addsub is
port ( addsub: in std_logic;
       x,y: in std_logic_vector(3 downto 0);
       s: out std_logic_vector(3 downto 0);
       cout, overflow: out std_logic);
end my_addsub;

architecture struct of my_addsub is
component fulladd
    port ( cin, x, y: in std_logic;
          s, cout: out std_logic);
end component;

signal c: std_logic_vector(4 downto 0);
signal yt: std_logic_vector(3 downto 0);

begin -- continued on next page
We copy what is in the entity of full_add.vhd

In order to use the file 'fulladd.vhd' into the top file, we need to declare it in the top file:
```
4-bit 2's complement Adder

Here, we:

- Insert the required extra circuitry (xor gates and I/O connections).
- Instantiate the full adders and interconnect them (using the `port map` statement)

```vhdl
-- continuation from previous page
  c(0) <= addsub; cout <= c(4);
  overflow <= c(4) xor c(3);

  yt(0) <= y(0) xor addsub; yt(1) <= y(1) xor addsub;
  yt(2) <= y(2) xor addsub; yt(3) <= y(3) xor addsub;

  f0: fulladd port map(cin=>c(0),x=>x(0),y=>yt(0),s=>s(0),cout=>c(1));
  f1: fulladd port map(cin=>c(1),x=>x(1),y=>yt(1),s=>s(1),cout=>c(2));
  f2: fulladd port map(cin=>c(2),x=>x(2),y=>yt(2),s=>s(2),cout=>c(3));
  f3: fulladd port map(cin=>c(3),x=>x(3),y=>yt(3),s=>s(3),cout=>c(4));

end struct;
```
4-bit 2’s complement Adder

- Use of ‘port map’ statement:
  
  ```
  port map (signal in full adder => signal in top file, ...)
  ```

- Instantiating and connecting the first full adder:

```vhdl
f0: fulladd port map (cin=>c(0), x=>x(0), y=>yt(0), s=>s(0), cout=>c(1));
```
4-bit 2's complement Adder

- Use of ‘port map’ statement:
  
  \texttt{port map (signal in full adder => signal in top file, ...)}

- Instantiating and connecting the second full adder:

  \texttt{f1: fulladd port map (cin=>c(1), x=>x(1), y=>yt(1), s=>s(1), cout=>c(2));}
4-bit 2’s complement Adder

- Use of ‘port map’ statement:
  
  ```
  port map (signal in full adder => signal in top file, ...)
  ```

- Instantiating and connecting the third full adder:

```
f2: fulladd port map(cin=>c(2), x=>x(2), y=>yt(2), s=>s(2), cout=>c(3));
```
4-bit 2’s complement Adder

- Use of ‘port map’ statement:
  
  ```
  port map (signal in full adder => signal in top file, ...)
  ```

- Instantiating and connecting the fourth full adder:

  ```
  f3: fulladd port map(cin=>c(3), x=>x(3), y=>yt(3), s=>s(3), cout=>c(4));
  ```
FOR-GENERATE Statement

- In the 4-bit adder example, if we wanted to use say 8 bits, we would need to instantiate 8 full adders and write 8 port map statements.
- Instantiating components can be a repetitive task, thus the for-generate statement is of great help here:

```vhdl
yt(0) <= y(0) xor addsub; yt(1) <= y(1) xor addsub;
yt(2) <= y(2) xor addsub; yt(3) <= y(3) xor addsub;

f0: fulladd port map (cin=>c(0), x=>x(0), y=>yt(0), s=>s(0), cout=>c(1));
f1: fulladd port map (cin=>c(1), x=>x(1), y=>yt(1), s=>s(1), cout=>c(2));
f2: fulladd port map (cin=>c(2), x=>x(2), y=>yt(2), s=>s(2), cout=>c(3));
f3: fulladd port map (cin=>c(3), x=>x(3), y=>yt(3), s=>s(3), cout=>c(4));
```

```
-- continuation from previous page
c(0) <= addsub; cout <= c(4);
overflow <= c(4) xor c(3);

gi: for i in 0 to 3 generate
    yt(i) <= y(i) xor addsub;
    fi: fulladd port map (cin=>c(i), x=>x(i), y=>yt(i), s=>s(i), cout=>c(i+1));
end generate;
end struct;
```
**INTRODUCTION TO PARAMETRIC CODING**

- N-bit adder/subtractor. We can choose the value of N in the entity.

- The architecture code is tweaked so as to make it parametric.

**Example:** Parametric N-bit adder/subtractor in 2’s complement:

- **my_addsub.zip:**
  - my_addsub.vhd
  - fulladd.vhd
  - tb_my_addsub.vhd
  - my_addsub.ucf

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity my_addsub is
  generic (N: INTEGER:= 4);
  port( addsub : in std_logic;
        x, y : in std_logic_vector (N-1 downto 0);
        s   : out std_logic_vector (N-1 downto 0);
        overflow : out std_logic;
        cout     : out std_logic);
end my_addsub;

architecture structure of my_addsub is
  component fulladd
    port( cin, x, y : in std_logic;
         s, cout : out std_logic);
  end component;

  signal c: std_logic_vector (N downto 0);
  signal yx: std_logic_vector (N-1 downto 0);
  begin
    c(0) <= addsub; cout <= c(N);
    overflow <= c(N) xor c(N-1);
    gi: for i in 0 to N-1 generate
        yx(i) <= y(i) xor addsub;
    fi: fulladd port map (cin=>c(i),x=>x(i),y=>yx(i),
                           s=>s(i),cout=>c(i+1));
  end generate;
end structure;
```
**INTRODUCTION TO PARAMETRIC CODING**

- **Example:** N-bit adder/subtractor.

- **Testbench:** Use of `for loop` inside the stimulus process to generate all possible input combinations.

- **Parametric Testbench:** It depends on N, which must match the hardware description

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- for conv_std_logic_vector

entity tb_my_addsub is
  generic (N: INTEGER:= 4); -- must match the HW description
end tb_my_addsub;

architecture structure of my_addsub is
  component my_addsub -- Do not do 'generic map' in testbench
    port( addsub : in std_logic;
        x, y : in std_logic_vector (N-1 downto 0);
        s    : out std_logic_vector (N-1 downto 0);
        overflow,cout : out std_logic);
  end component;

  -- Inputs
  signal addsub: std_logic:='0';
  signal x,y: std_logic_vector (N-1 downto 0):= others => '0';

  -- Outputs
  signal overflow, cout: std_logic;
  signal s: std_logic_vector (N-1 downto 0);

begin
  uut: my_addsub port map (addsub, x, y, s, overflow, cout);
  st: process
  begin
    wait for 100 ns;
    addsub <= '0'; -- Pick '1' to test subtraction
    gi: for i in 0 to 2**N-1 loop
      y <= conv_std_logic_vector(i,N); wait for 10 ns;
      gj: for j in 0 to 2**N-1 loop
        x <= conv_std_logic_vector(j,N); wait for 10 ns;
        y <= std_logic_vector(3 downto 0) => '0';
      end loop;
      wait;
    end loop;
    wait;
  end process;
end;
```

Daniel Llamocca
Example: 4-bit array multiplier

- Interesting example of structural description. By connecting full adders and AND gates, we build the array multiplier.

- We parameterize the circuit with ‘if-generate’, ‘for-generate’, and the use of arrays in VHDL.

- Testbench: we use conv_std_logic_vector to specify input values

Parametric N-bit array multiplier:

- my_mult.zip:
  - my_mult.vhd.vhd, fulladd.vhd
  - tb_my_mult.vhd, my_mult.ucf

Daniel Llamocca
Example: Arithmetic Logic Unit

- This circuit executes two types of operations: logic (or bit-wise) and arithmetic.
- The arithmetic unit and the logic unit are described in different VHDL files.
- The arithmetic unit relies heavily on the parametric adder/subtractor unit.
- The VHDL code is parameterized so as to allow for two N-bit operands.
- The ‘sel’ inputs selects the operation to be carried on (as per the table).

- my_alu.zip: my_alu.vhd, my_alu_arith.vhd, my_alu_logic.vhd, my_addsub.vhd, fulladd.vhd
  tb_my_alu.vhd

<table>
<thead>
<tr>
<th>sel</th>
<th>Operation</th>
<th>Function</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>y &lt;= a</td>
<td>Transfer 'a'</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>y &lt;= a + 1</td>
<td>Increment 'a'</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>y &lt;= a - 1</td>
<td>Decrement 'a'</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>y &lt;= b</td>
<td>Transfer 'b'</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>y &lt;= b + 1</td>
<td>Increment 'b'</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>y &lt;= b - 1</td>
<td>Decrement 'b'</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>y &lt;= a + b</td>
<td>Add 'a' and 'b'</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>y &lt;= a - b</td>
<td>Subtract 'b' from 'a'</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>y &lt;= NOT a</td>
<td>Complement 'a'</td>
<td>Logic</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>y &lt;= NOT b</td>
<td>Complement 'b'</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>y &lt;= a AND b</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>y &lt;= a OR b</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>y &lt;= a NAND b</td>
<td>NAND</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>y &lt;= a NOR b</td>
<td>NOR</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>y &lt;= a XOR b</td>
<td>XOR</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>y &lt;= a XNOR b</td>
<td>XNOR</td>
<td></td>
</tr>
</tbody>
</table>
Example: 6-to-6 LUT

- The LUT contents (64 bytes) are specified as a set of 6 parameters. These 6 parameters are 'generics' in the VHDL code.
- Note that if the parameters are modified, we will get a different circuit that needs to be re-synthesized. In other words, the LUT contents are NOT inputs to the circuit.

my6to6LUT.zip: my6to6LUT.vhd, my6to1LUT.vhd, my5to1LUT.vhd, my4to1LUT.vhd, tb_my6to6LUT.vhd, my6to6LUT.ucf