University of the Philippines Los Baños
College of Arts and Sciences
Institute of Computer Science
CMSC 132 - Computer Architecture
First Semester AY 2022-2023

Laboratory Topic 03: Combinational Circuits

RNC Recario {rcrecario@up.edu.ph}

Learning Outcomes:

At the end of the laboratory session, the student is expected to:

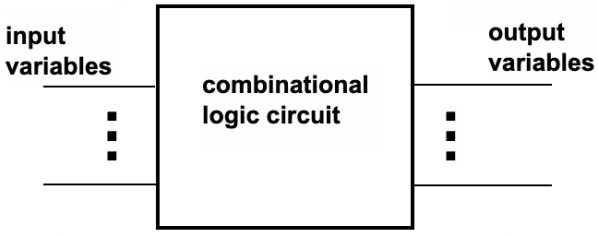
- Define what a combinational circuit is
- Explain how components and instantiation is done for complex designs
- Implement a combinational circuit with several components

----------**Topic Proper**----------

What is a combinational circuit?

A **combinational circuit** "comprises of logic gates whose outputs at any time are determined directly from the present combination of inputs without any regard to previous inputs" [1].

A block diagram of a combinational circuit can be summarized by the one shown below.



How are combinational circuits implemented in VHDL?

What has been done in the previous topic is essentially how we implement a combinational circuit. However, the example and the exercise were basic. What if the design (i.e. circuit) is complex and big? How will you implement it?

We now introduce the concept of components and instantiation. Both concepts are already used in the previous topic but were not given details.

What is a component/ VHDL component?

A **component** represents "an entity/architecture pair". It "specifies a subsystem, which can be instantiated in another architecture leading to a hierarchical specification" [2]. You can think of components as functions that can be reused by other functions.

To better demonstrate the concept, let us use the example of a 3-input (*i0, i1, and i2*) AND gate as our combinational circuit. We know that we can simply declare it as a direct expression similar to the one below:

```
andgate_result <= i0 and i1 and i2;
```

But suppose we insist that the implementation use components? In such a case, we break the design even further. An example is to use a 2-input AND gate and reuse that AND gate to accept the result of the first two signals (e.g., i0 and i1) and perform an AND with the third one (i2). The diagram below will help you visualize how the two-input implementation is intended to be done.

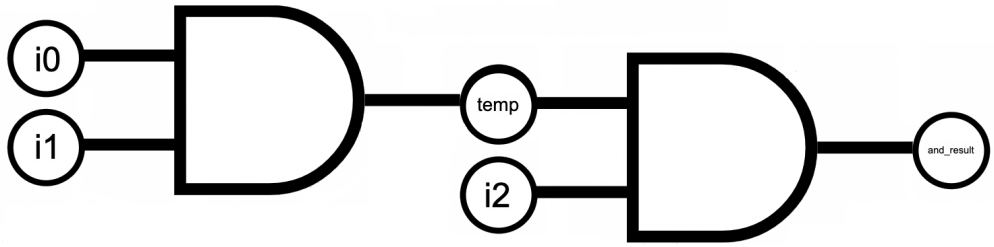


Figure 1. A detailed representation of a 3-input AND gate implemented using two AND gates.

The initial inputs are i0 and i1. The result from the first AND gate is represented by temp which is ANDed to i2. The final result is represented by the signal and_result. A more general diagram to show the above circuit is to simply show a block diagram. An example is shown below:

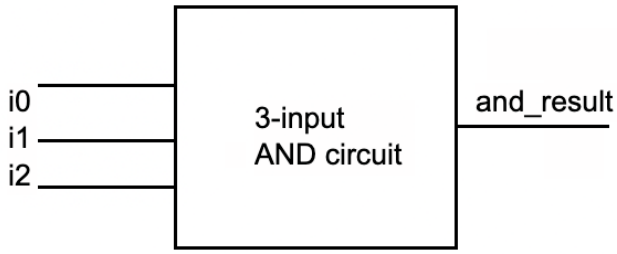


Figure 2. Block diagram of the 3-input AND gate

To implement the components, we need to identify the smaller parts of the 3-input AND gate. We know from the drawing that there are only two AND gates. And both have similar

implementation but with different input and output signals. Hence , we will implement an AND gate first. The code below demonstrates this move.

andgate2.vhdl

```
entity andgate2 is
  port (i0, i1: in bit; andresult : out bit);
end entity;

architecture rtl of andgate2 is
begin
        andresult <= (i0 and i1);
end architecture;
```

As you can see, we implement simply using the AND operation for i0 and i1. This AND gate, however, will be reused and thus, the inputs i0 and i1 (declared in the andgate2 entity) will represent the i0 and i1 from our 3-input AND gate, and temp and i2. Since there is only one type of component (only AND gate which will be reused twice), we can already create our main design file which is named andgate3. The code below shows the implementation of andgate3.

andgate3.vhdl

```
1   entity andgate3 is
2     port (i_0, i_1, i_2 : in bit; and_result : out bit);
3   end entity;
4
5   architecture rtl of andgate3 is
6           signal temp: bit;
7           component andgate2 is
8                   port (i0, i1 : in bit; andresult : out bit);
9           end component;
10  begin
11          and_gate_0: andgate2 port map(i0 => i_0, i1 => i_1, andresult => temp);
12          and_gate_1: andgate2 port map(i0 => temp, i1 => i_2, andresult => and_result);
13  end architecture;
```

As always, the entity name should be descriptive (in the example above, you can further improve the naming instead of andgate3 which I used as a shortcut for an AND gate with 3 inputs).

In lines 7-10, you see that there is the declaration of the component similar to what has been done in topic 2. All components of the current design file should be declared in this area between the `architecture` declaration and the `begin` body. There is only one type of component to be used and declared - the AND gate. While it is true that the 2-input AND gate will be reused twice, there is no need to declare the component twice. Instead, the component will be instantiated twice in the architecture body.

Lines 11 and 12 show the instantiation and the use of the AND gate we declared as a component. You will notice that the instantiation follows a format. The first thing declared is the instantiation variable (`and_gate_0` and `and_gate_1`) which you can name freely like a variable (note that you cannot use keywords, redeclare variable name, and use special symbols). Following the instantiation is the colon and the name of the component you will use. Following the component name is the `port map` statement which will map the signals from the component to the signals declared in the design file. Refer to figure 1 to see how the vhdl file conforms.

After creating the "main" design file, you can already write your test bench file for the "main" design file. The code below shows that.

andgate3_tb.vhdl

```
1   entity andgate3_tb is
2   end andgate3_tb;
3
4       architecture behav of andgate3_tb is
5         -- Declaration of the component that will be instantiated.
6         component andgate3
7           port (i_0, i_1, i_2 : in bit; and_result : out bit);
8         end component;
9         -- Specifies which entity is bound with the component.
10        for andgate3_0: andgate3 use entity work.andgate3;
11        signal i0, i1, i2, and_result : bit;
12      begin
13        -- Component instantiation.
14        andgate3_0: andgate3 port map (i_0 => i0, i_1 => i1, i_2 => i2,
15                                and_result => and_result);
16
17        -- This process does the real job.
18        process
```

```
19          type pattern_type is record
20            -- The inputs of the and gate.
21            i0, i1, i2 : bit;
22            -- The expected outputs of the and gate.
23            and_result : bit;
24          end record;
25          -- The patterns to apply.
26          type pattern_array is array (natural range <>) of pattern_type;
27          constant patterns : pattern_array :=
28           (('0', '0', '0', '0'),
29            ('0', '0', '1', '0'),
30            ('0', '1', '0', '0'),
31            ('0', '1', '1', '0'),
32            ('1', '0', '0', '0'),
33            ('1', '0', '1', '0'),
34            ('1', '1', '0', '0'),
35            ('1', '1', '1', '1'));
36        begin
37          -- Check each pattern.
38          for i in patterns'range loop
39            -- Set the inputs.
40            i0 <= patterns(i).i0;
41            i1 <= patterns(i).i1;
42            i2 <= patterns(i).i2;
43            -- Wait for the results.
44            wait for 1 ns;
45            -- Check the outputs.
46            assert and_result = patterns(i).and_result
47              report "bad and value" severity error;
48          end loop;
49          assert false report "end of test" severity note;
50          -- Wait forever; this will finish the simulation.
51          wait;
52        end process;
53      end behav;
```

The idea in the test bench is similar to what we have done in the previous topic. Note also the pattern in lines 28-35 which corresponds to the 3-input signal and the output. This pattern also corresponds to the truth table for the 3-input AND gate which obviously generates a '1' (HIGH) when all the input signals are '1' (HIGH).

After creating the component file (andgate2.vhdl), the design file (andgate3.vhdl) and the test bench, you can compile them all similar to what you have done in the past. Or, a shortcut: just compile the test bench:

```
ghdl -a andgate3_tb.vhdl
ghdl -e andgate3_tb
ghdl -r andgate3_tb
```

Note that all the three files (andgate2.vhdl, andgate3.vhdl, and andgate3_tb.vhdl) should be in the same directory (aka library "work").

Example 2: Bitwise OR operation.
This time, another example. How about a 3-bit input signal (bitwise) OR operation? How will you implement it?

Let's create the detailed representation and the block diagram respectively:

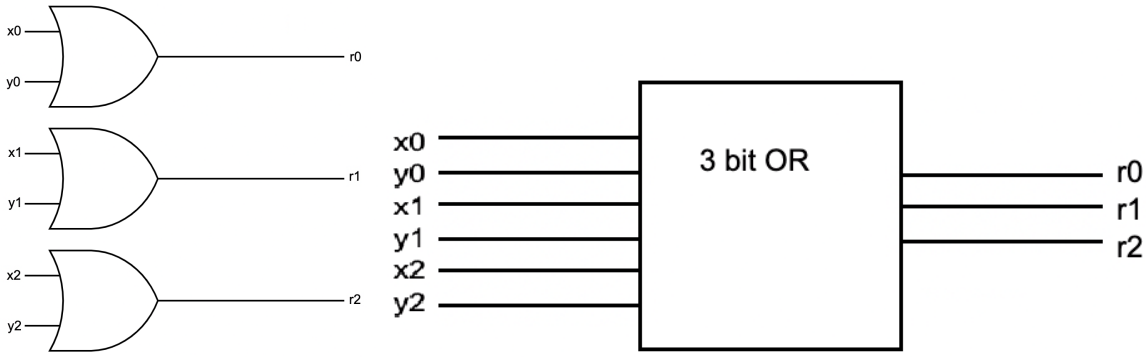


Figure 3. Left: A detailed representation of a 3-bit input for an OR gate for a bitwise OR operation. Right: Block representation of a 3-bit input for an OR gate for a bitwise OR operation.

The basic building component obviously is an OR gate which is implemented in vhdl.

orgate.vhdl

```
entity orgate is
  port (i0, i1: in bit; orresult : out bit);
end entity;

architecture rtl of orgate is
begin
        orresult <= (i0 or i1);
end architecture;
```

The main design file would look like this:

bitwise_or.vhdl

```
1   entity bitwise_or is
2     port (x0, x1, x2, y0, y1, y2: in bit; r0, r1, r2 : out bit);
3   end entity;
4
5   architecture rtl of bitwise_or is
6   component orgate is
7     port (i0, i1: in bit; orresult : out bit);
8   end component;
9   begin
10          orgate_0: orgate port map(i0 => x0, i1 => y0, orresult => r0);
11          orgate_1: orgate port map(i0 => x1, i1 => y1, orresult => r1);
12          orgate_2: orgate port map(i0 => x2, i1 => y2, orresult => r2);
13  end architecture;
```

As you can see from the code, in order to implement the bitwise or problem, we have to accept six (6) signals representing a pair from two 3-bit signals x and y. The result or output of course is also 3 bits.

Lastly, we show how the test bench is created. In the interest of brevity, we limited the values to test.

bitwise_or_tb.vhdl

```
1   -- A testbench has no ports.
2      entity bitwise_or_tb is
3      end bitwise_or_tb;
```

```
4
5       architecture behav of bitwise_or_tb is
6         -- Declaration of the component that will be instantiated.
7         component bitwise_or
8           port (x0, x1, x2, y0, y1, y2: in bit; r0, r1, r2 : out bit);
9         end component;
10        -- Specifies which entity is bound with the component.
11        for bitwise_or_0: bitwise_or use entity work.bitwise_or;
12        signal x0, x1, x2, y0, y1, y2, r0, r1, r2 : bit;
13      begin
14        -- Component instantiation.
15        bitwise_or_0: bitwise_or port map (x0 => x0, x1 => x1, x2 => x2, y0 => y0, y1 => y1,
16    y2 => y2,
17         r0 => r0, r1 => r1, r2 => r2);
18
19        -- This process does the real job.
20        process
21          type patterntype is record
22            -- The inputs of the and gate.
23            x0, x1, x2, y0, y1, y2 : bit;
24            -- The expected outputs of the and gate.
25            r0, r1, r2 : bit;
26          end record;
27          -- The patterns to apply.
28          type patternarray is array (natural range <>) of patterntype;
29
30          --note: this truth table is incomplete and needs to be extended
31          constant patterns : patternarray :=
32            (('0','0','0','0','0','0','0','0','0','0'),
33             ('0','0','0','0','0','1','0','0','1'),
34             ('0','0','0','0','1','0','0','1','0'),
35             ('0','0','0','0','1','1','0','1','1'));
36        begin
37          -- Check each pattern.
38          for i in patterns'range loop
39            -- Set the inputs.
40            x0 <= patterns(i).x0;
41            x1 <= patterns(i).x1;
42            x2 <= patterns(i).x2;
43            y0 <= patterns(i).y0;
44            y1 <= patterns(i).y1;
```

```
45          y2 <= patterns(i).y2;
46          --  Wait for the results.
47          wait for 1 ns;
48          --  Check the outputs.
49          assert r0 = patterns(i).r0
50            report "bad r0 value" severity error;
51          assert r1 = patterns(i).r1
52            report "bad r0 value" severity error;
53          assert r2 = patterns(i).r2
54            report "bad r0 value" severity error;
55        end loop;
56        assert false report "end of test" severity note;
57        --  Wait forever; this will finish the simulation.
58        wait;
59      end process;
60    end behav;
```

Similarly, in order to test if the above-mentioned files work, you have to "compile" and run them separately or just simply run and compile the test bench (Note that all files should be in the same directory. If there are updates on the code, affected components should be reanalyzed first).

References:

[1] https://www.javatpoint.com/combinational-circuits
[2] https://peterfab.com/ref/vhdl/vhdl_renerta/mobile/source/vhd00016.htm