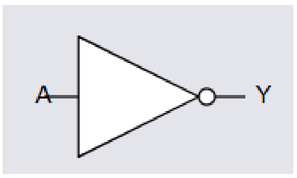
1)Introducing Logic Gates

1.1) The 3 Most Fundamental Gates (And the Buffer)

1) The NOT Gate



This can also be expressed as Y = Not A.

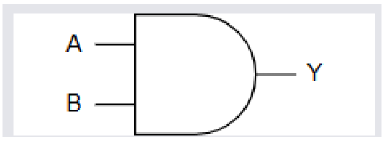
It has 1 input, and therefore $2^1 = 2$ combinations of inputs. The combinations of inputs and their resulting outputs can be organized through a through table. Here's the truth table for the not gate.

A	Y
0	1
1	0

In the binary world, 0 means off and 1 means on. The truth table can be interpreted in this manner.
- When A(input) is 0, Y(output) is 1.
- When A is 1, Y is 0.

In basic English, this means:
- When A is off, Y is on.
- When A is on, Y is off.

2) The AND Gate



This can be expressed as $Y = A \text{ and } B$

or $Y = A * B$ (A times B).

We have 2 inputs (A,B) and therefore $2^2 = 4$ combinations of inputs. The output is only when A and B are both 1. $B * 0$ or $A * 0$ will always be 0, so Y is off if any inputs are off. In order to avoid memorization, consider this.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Suppose we have 4 variables (A,B, C, D), whose combinations can be written as a 4 bit number. Like when all variables are 0, we have "0000". These numbers can either be 1 or 0 and we have 4 of them.This means we have $2 * 2 * 2 * 2$ or 2^4 possible combinations. How are creating a truth table without repeating any combinations?

Remember place values from elementary and middle school? Let's look at a 4 digit number in decimal (base 10). This means that it can be represented with 10^n and is made of 9 digits (0,1,2,3,4,5,6,7,8,9). Let's look at the number 4268.

- 4 is in the thousandths or 10^3 spot.
- 2 is in the hundredths or 10^2 spot.
- 6 is in the tenths or 10^1 spot.
- 8 is in the ones or 10^0 spot.

By combining them as a sum, we can represent 4268 as $4 * 10^3 + 2 * 10^2 + 6 * 10^1 + 8 * 10^0$. By adding all these numbers, we obtain 4268. This shows both forms are equivalent.

This approach can be applied to various number systems, including binary (base 2). Confused or Interested in why? Observe!

Let's look at a 4 binary digit like 1001. Let's start with the digit that is the most left.

- 1 is in the 2^0 or 1 spot.
- 0 is in the 2^1 or 2 spot.
- 0 is in the 2^2 or 4 spot.
- 1 is in the 2^3 or 8 spot.

This number can be represent as $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$ or $8 + 0 + 0 + 1$. By adding them all up, we get 9.

So why did we take a detour about number systems when discussing how to make truth table without memorization?

Let's think of any 4 digits composed of the 4 variables A, B, C, D. The 4 digit number looks like ABCD.

A is in the 2^3 or 8 spot.

B is in the 2^2 or 4 spot.

C is in the 2^1 or 2 spot.

D is in the 2^0 or 1 spot.

A is in the spot that's worth the most while D is in the spot that's worth the least. Another way of saying this is that A is in the most significant bit (MSB) and D is in the least significant bit (LSB). On the truth table, the MSB is placed on the most right while the LSB is placed on the most left (excluding outputs like Y).

Binary numbers are also called "switching algebra". On a truth table, a variable's spot is the amounts of 0 before its switches to an identical number of 1s. This process repeats till the table is filled.

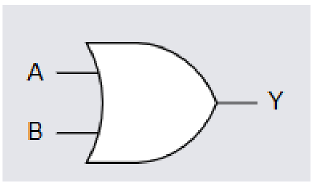
- D has a spot value of 1. So we start with 1 zero, then switch to one 1, and then 1 zero.
- C has a spot value of 2. So we start with 2 zero, then switch to two 1, and then 2 zero.
- B has a spot value of 4. So we start with 4 zero, then switch to four 1, and then 4 zero.
- A has a spot value of 8. So we start with 8 zero, then switch to eight 1, and then 8 zero.

This iterating or repetition repeats until the truth table is filled. Here's how an truth table of 4 variables and no output look like.

A	B	C	D
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Focus only in the pattern, not the values itself.

3) The OR Gate



This can be expressed as Y = A OR B

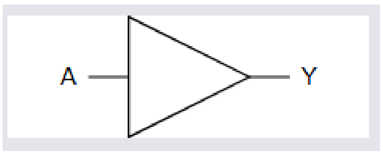
or Y = A + B (A plus B)

We have 2 inputs and $2^2 = 4$ combinations of these inputs.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Y is 1 if any of its inputs (A, B, or All) are on. Y will be on if any of its input (switches) are on. If none of the inputs are on, Y will be off.

4) The BUFFER Gate



A	Y
0	0
1	1

If A is 1, Y is 1. If A is 0, then Y is 0.

If A is on, Y is on. If A is off, Y is off.

1.2) VHDL Implementation of the 4 Basic Gates

——— Code Between these marks represent code on software capable of running an Hardware Description Language (HDL) like VHDL and Verilog. I like VHDL more. I'll be using Vivado as a software since it's free.———

Just like C (programming language) and HTML (Web Description Language), you need to write a manual declaration. That's needed for the software to recognize the language that will be used. An perk of VHDL is that spacing and capitalization doesn't matter. Don't worry about these "issues" when debugging.

Here's the declaration in VHDL. Later, you can include other packages.

——

```
library ieee;

use ieee.std_logic_1164.all;
```

——

Then we name our file. Following that, we list our input and outputs. That creates a port-map.It's called that since it maps out your switches, buttons, or any input/output device.

file_name is the name of your file. It's best if your file_name listed in the code is the same as the name of the vhdl file. If the file is called and_gate.vhdl, its filename you probably be and_gate.

In VHDL, you list all inputs and outputs and classify them as such. There are more than basic inputs and outputs but it's not relevant right now.

In the port map, we don't need the last concluding semicolon. It's already included.

——

```
entity file_name is

port(

variable 1, variable 2 : in std_logic ;

variable 3 : out std_logic

);

end entity;
```

——

Then we have the last section where the output and external wires are defined. Our output us equal to logic operations being performed on the inputs. Instead of an equal sign (=) or double equal sign (==), we use ‘<=’.

We also define logic gates’ behavior here and architecture type here.

——

```
architecture arch_type of file_name is begin

variable 3 <= variable 1 [logic expression] variable 2

end arch_type
```

——

Here’s the and gate. 1 bit inputs and 1 bit outputs.

———

———

———

——

```
library ieee;

use ieee.std_logic_1164.all;

entity and_gate is

port(a,b : in std_logic;

y : out std_logic

);

end entity;

architecture behavioral of and_gate is

begin

y <= a and b;

end behavioral;
```

——

Here’s the or gate. 1 bit inputs and 1 bit outputs.

———

———

———

——

```
library ieee;

use ieee.std_logic_1164.all;

entity or_gate is
```

```
port(
a,b : in std_logic;
y : out std_logic
);
end entity;
architecture behavioral of or_gate is
begin
y <= a or b;
end behavioral;
```

——

Here’s the buffer gate. 1 bit input and 1 bit output.

———

———

———

——

```
library ieee;
use ieee.std_logic_1164.all
entity buffer_gate is
port(
a : in std_logic;
y : out std_logic
);
end entity;
architecture behavioral of buffer_gate is
begin
y <= a;
end behavioral;
```

——

Here’s the not gate. 1 bit inputs and 1 bit outputs.

———

———

———

——

——

```
library ieee;
use ieee.std_logic_1164.all;
entity not_gate is
port (
a : in std_logic;
y : out std_logic
);
end entity;
architecture behavioral of not_gate is
begin
y <= not a;
end behavioral;
```

1.3) Timing Diagrams and Test Benches for 4 Basic Gates

Timing diagrams are visual ways to see and perform binary calculations. There, you see numerous shapes that exist between a starting time and finishing time. All shapes under the same column region are related. Test-benches allow you to create these shapes as well as assigning them values.

Let’s look at the OR gate.

Here’s its test bench. Recall it has 2 inputs and 1 output. The and gate also has the same amount of inputs and outputs. The test bench allows us to plug input values into the timing diagrams. Those timing diagrams are the result of the VHDL code.

Let’s look at the test-bench for the OR gate.

```
library ieee;
use ieee.std_logic_1164.all;
entity tb_or_gate is
end entity tb_or_gate;
architecture sim of tb_or_gate is
signal a_in : std_logic := ‘0’;
signal b_in : std_logic := ‘0’;
signal y_out : std_logic;
begin
uut : entity work.or_gate
port map (
a => a_in,
b => b_in,
y => y_out
);
stim_proc : process
begin
a_in <= ‘0’; b_in <= ‘0’;
wait for 10 ns;
a_in <= ‘0’; b_in <= ‘1’;
wait for 10 ns;
a_in <= ‘1’; b_in <= ‘0’;
wait for 10 ns;
a_in <= ‘1’; b_in <= ‘1’;
wait for 10 ns;
wait;
end process;
end architecture sim;
```

Let’s look at:

- entity tb_or_gate is
- end entity tb_or_gate;

Here we define our test-bench but not its inputs or outputs.

Then we define the inputs and outputs using external signal wires.

- architecture sim of tb_or_gate is
- signal a_in : std_logic := ‘0’;
- signal b_in : std_logic := ‘0’;
- signal y_out : std_logic;

The := means we start at the value 0.

Third, we assign a signal wire to its equivalent internal input/output. That is done on the port map section in the test bench.

- External signal a is connected to a input
- External signal b is connected to b input
- External signal y is connected to y output

Finally, we break up timing diagrams in blocks of time. Within each blocks, we have specific inputs that cause specific outputs.

Below is how its done in VHDL.

- stim_proc : process
- begin
- a_in <= ‘0’; b_in <= ‘0’;
- wait for 10 ns;
- a_in <= ‘0’; b_in <= ‘1’;
- wait for 10 ns;
- a_in <= ‘1’; b_in <= ‘0’;
- wait for 10 ns;
- a_in <= ‘1’; b_in <= ‘1’;
- wait for 10 ns;
- wait;
- end process;

With this test-bench in mind, here’s our timing diagram.

1.4) Creating All Other Gates From And, Not, and Or Gate

1.5) Creating And, Not, and Or Through Nand and Nor Gates

1.6) Creating All Logic Gates from Nand

1.7) Creating All Logic Gates From Nor