Combinational Logic Design

This lecture reviews the design of combination logic and introduces the use of VHDL for logic design.
After this lecture you should be able to convert a text description of a combinational logic circuit into three forms: (1) a truth table, (2) a sum of products boolean equation, and (3) a VHDL entity/architecture description.

Logical Variables and Boolean Algebra

A logic variable can take on one of two values, typically called true and false (T and F). With positive-true logic true values are represented by a high (H) voltage. With negative-true logic true values are represented by a low (L) voltage. Variables using negative logic are usually denoted by placing a bar over the name ($\overline{\mathrm{B}}$), or an asterisk after the variable name (B^*).

Warning: We will use or 1 and 0 to represent *truth values* rather than voltage levels. However, some authors use 1 and 0 to represent voltage levels instead. This can be very confusing. Also note that an overbar is sometimes used to indicate a logical complement operation rather than a negative-true signal.

Combinational Logic

A combinational logic circuit is one where the output is a function of the current input only. A combinational logic circuit can be represented as:

- *a truth table* that shows the output values for each possible combination of input values,
- *a boolean equation* that defines the value of each output variable as a function of the input variables, or
- *a schematic* that shows a connection of hardware logic gates (and possibly other devices) that implement the circuit.

Truth Tables

For example, the truth table for a circuit with an output that shows if its three inputs have an even number of 1's (even parity) would be:

a	b	c	p
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Sum of Products Form

From the truth table we can obtain an expression for each output as a function of the input variables.

The simplest method is to write a *sum of products* expression. Each term in the sum corresponds to one line of the truth table for which the desired output variable is true (1). The term is the product of the each input variable (if that variable is 1) or its complement (if that variable is 0). Each such term is called a *minterm* and such an equation is said to be in *canonical* form.

For example, the variable p above takes on a value of 1 in four lines (the first, fourth, sixth and seventh lines) so there would be four terms. The first term corresponds to the case where the input variables are $a = 0$, $b = 0$ and $c = 0$. So the term is $\overline{\mathrm{a}}\overline{\mathrm{b}}\overline{\mathrm{c}}$. Note that this product will only be true when a, b and c have the desired values, that is, only for the specific combination of inputs on the third line.

If we form similar terms for the other lines where the desired output variable takes on the value one and then sum all these terms we will have an expression that will evaluate to one only when required and will evaluate to zero in all other cases.

Exercise: Write out the sum-of-products equation for p. Evaluate the expression for the first two lines in the table.

Common Combinational Logic Circuits

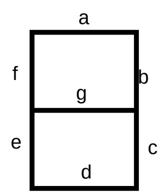
In addition to the standard logic functions (NAND, OR, etc) some combinational logic functions are widely used and are often available as SSI ICs and as library blocks for ASIC design:

- a *decoder* is a circuit with N inputs and 2^N outputs. The output selected by the input bits (treated as a binary number) is set true and the other outputs are false. This circuit is often used for address decoding.
- a *priority encoder* does the inverse operation. The N output bits represent the number of the (highest numbered) input line.
- a *multiplexer* copies the value of one of 2^N inputs to a single output. The input is selected by an N-bit input.
- a *demultiplexer* does the inverse operation and copies one input to one of 2^N outputs.
- *adders*, *comparators* and *ALUs* (arithmetic logic units) implement the basic arithmetic and logic functions
- *drivers* and *buffers* provide high current, tri-state or open collector (OC) outputs

Exercise: Write out the truth table and the canonical sum-of-products expression for a 2-to-1 multiplexer.

Example: 7-segment display driver

LED numeric displays typically use seven segments labeled ‘a’ through ‘g’ to display a digit between 0 to 9:



This example shows the design of a circuit that converts a 2-bit number into seven outputs that turn the segments on and off to show numbers between 0 and 3. We use the variables A and B for the two input bits and a to g for the seven outputs. We can build up a truth table for this function as follows:

B	A	a	b	c	d	e	f	g
0	0	1	1	1	1	1	1	0
0	1	0	1	1	0	0	0	0
1	0	1	1	0	1	1	0	1
1	1	1	1	1	1	0	0	1

From the truth table we can then write out the sum of products expressions for each of the outputs:

$$\begin{aligned}
a &= \overline{A}\overline{B} + \overline{A}B + AB \\
b &= 1 \\
c &= \overline{A}\overline{B} + A\overline{B} + AB \\
d &= \overline{A}\overline{B} + \overline{A}B + AB \\
e &= \overline{A}\overline{B} + \overline{A}B \\
f &= \overline{A}\overline{B} \\
g &= \overline{A}B + AB
\end{aligned}$$

Exercise: Draw a schematic of a circuit that implements the above logic function.

Combinational Logic Design with VHDL

VHDL is a Very-complex[1] Hardware Description Language that we will use to design logic circuits.

Example 1 - Introduction

Let’s start with a simple example – a type of circuit called example1 that has one output signal (`c`) that is the AND of two input signals (`a` and `b`). The file `example1.vhd` contains the following VHDL description:

```
-- example 1: An AND gate

entity example1 is port (
   a, b: in bit ;
   c: out bit ) ;
end example1 ;

architecture rtl of example1 is
begin
   process(a,b)
   begin
      c <= a and b ;
   end process ;
end rtl ;
```

First some observations on VHDL syntax:

[1]Actually, the V stands for VHSIC. VHSIC stands for Very High Speed IC.

- VHDL is case-insensitive. There are many capitalization styles. I prefer all lower-case. You may use whichever style you wish as long as you are consistent.

- Everything following two dashes "`--`" on a line is a comment and is ignored.

- Statements can be split across any number of lines. A semicolon ends each statement. Indentation styles vary but an "end" should be indented the same as its corresponding "begin"

- Entity and signal names begin with a letter followed by letters, digits or underscore ("_") characters.

A VHDL description has two parts: an entity part and an architecture part. The entity part defines the input and output signals for the device or "entity" being designed while the architecture part describes the behaviour of the entity.

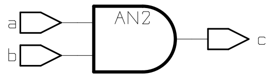
Each architecture is made up of one or more processes, all of which "execute[2]" at the same time (*concurrently*). Within each process, statements are executed one after the other (*sequentially*).

The `(a,b)` after `process` is called the "sensitivity list" and should include all the signals that might affect the value(s) computed in the process.

The single statement in this example's single process is a signal assignment that assigns the value of an expression to the output signal `c`. Expressions involving signals of type bit can use the logical operators `and`, `nand`, `or`, `nor`, `xor`, `xnor`, and `not`. Parentheses can be used to force evaluation in a certain order.

To ensure that we'll end up with a simple combinational circuit each output signal should be assigned a value exactly once in the architecture.

From this VHDL description a program called a logic synthesizer can design a circuit that has the required functionality. In this case it's not too surprising that the result is the following circuit:



[2]The resulting hardware doesn't actually "execute" but this point of view is useful when using VHDL for simulation.

Example 2 - if/then/else

Within a process we can use other programming language constructs such as `if/then/else` and `case` statements.

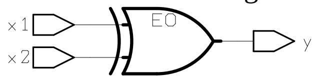
The next example uses an `if/then/else` statement in a process:

```
-- example 2: XOR using if/then/else

entity example2 is port (
        x1, x2: in bit ;
        y: out bit ) ;
end example2 ;

architecture rtl of example2 is
begin
   process(x1,x2)
   begin
      if x1 /= x2
      then
         y <= '1' ;
      else
         y <= '0' ;
      end if ;
   end process ;
end rtl ;
```

which synthesizes to the following:



Relational operators that can be used in expressions include `=` (equal), `/=` (not equal), `>`, `<`, `>=`, and `<=`. The result of relational operators is of type boolean.

The logical operators (e.g. `and`) can also be applied to values of type boolean. However, note that in general there is no automatic type conversion in VHDL. For example, boolean values cannot be assigned to bit values and integers cannot be assigned to bit_vectors.

Exercise: What schematic would you expect if the `/=` were replaced with = ?

Example 3 - Vectors

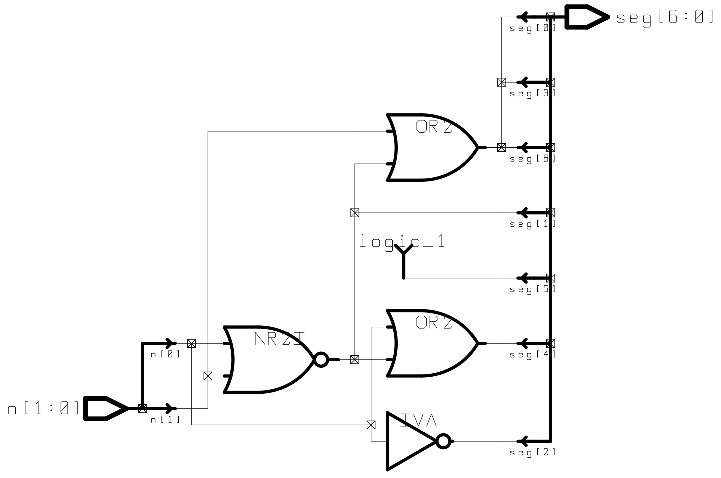
VHDL also allows signals of type `bit_vector` which are one-dimensional arrays of bits. The next example is a VHDL description of the 7-segment LED driver and demonstrates the `case` statement and bit_vectors.

```
-- example 3: 7-segment LED driver for
--    2-bit input values
```

```
entity example3 is port (
   n: in bit_vector (1 downto 0) ;
   seg: out bit_vector (6 downto 0) ) ;
end example3 ;

architecture rtl of example3 is
begin
   process(n)
   begin
      case n is
         when "00" => seg <= "1111111" ;
         when "01" => seg <= "0110000" ;
         when "10" => seg <= "1101101" ;
         when "11" => seg <= "1111001" ;
      end case ;
   end process ;
end rtl ;
```

which synthesizes to:



The indices of bit_vectors can be declared to have increasing (`upto`) or decreasing (`downto`) values. `bit_vector` constants are formed by enclosing an ordered sequence of bit values in double quotes.

Exercise: If `x` is declared as `bit_vector (0 upto 3)` and in a process the assignment `x<="0011"` is made, what is the value of `x(3)`? What if `x` had been declared as `bit_vector (3 downto 0)`?

Substrings of vectors can be extracted by specifying a range in the index expression and vectors can be concatenated using the `&` operator. For example `x <= x(6 downto 0) & '0'` would shift the 8-bit value `x` left by 1 bit.

The logical operators (e.g. `and`) can be applied to bit_vectors and operate on a bit-by-bit basis. The `=` and `/=` operators can also be used and evaluate to true only if all elements are the same.

Exercise: Write a VHDL entity and architecture for the 2-to-1 multiplexer designed in the previous exercise. Use a case statement.

Exercise: Re-write example 3 so that the architecture uses only signal assignments.