Combinational Logic Design with VHDL

This lecture reviews the design of combinational logic and introduces the use of VHDL to design combinational logic. After this lecture you should be able to:

- *convert an informal description of the behaviour of a combinational logic circuit into a truth table, a sum of products boolean equation and a schematic,*
- *convert an informal description of a combinational logic circuit into a VHDL entity and architecture,*

Logic Variables and Signals

A logic variable can take on one of two values, typically called true and false (T and F). With *active-high* logic true values are represented by a high (H) voltage. With *active-low* logic true values are represented by a low (L) voltage. Variables using negative logic are usually denoted by placing a bar over the name ($\overline{\text{B}}$), 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 people use 1 and 0 to represent voltage levels instead. This can be very confusing.

Exercise 1: A chip has an input labelled $\overline{\text{OE}}$ that is used to turn on ("enable") its output. Is this input an active-high or active-low signal? Will the output be enabled if the input is high? Will the output be enabled if the input is 1?

To add to potential sources of confusion, an overbar is sometimes used to indicate a logical complement operation rather than a negative-true signal. The only way to tell the difference is from the context.

Combinational Logic

A combinational logic circuit is one where the output is a function only of the current input – not of any past inputs. 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

Exercise 2: Fill in the last two rows.

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 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 $\overline{a}\,\overline{b}\,\overline{c}$. 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 first line.

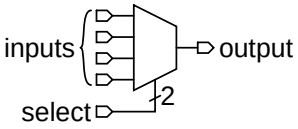
If we form similar terms for the other lines where the desired output variable takes on the value one and then sum all these terms we will have an expression that will evaluate to one only when required and will evaluate to zero in all other cases.

Exercise 3: Write out the sum-of-products equation for *p*. Evaluate the expression for the first two lines in the table.

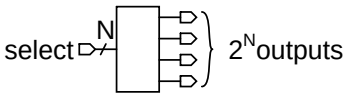
Common Combinational Logic Functions

In addition to the standard logic functions (AND, OR, NOT, XOR, NAND, etc) some combinational logic functions that are widely used include:

- a *multiplexer* is probably the most useful combinational logic circuit. It copies the value of one of 2^N inputs to a single output. The input is selected by an N-bit input.

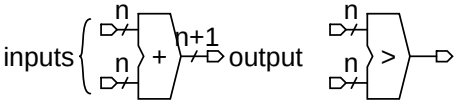


- a *decoder* is a circuit with N inputs and 2^N outputs. The input is treated as a binary number and the output selected by the value of the input is set true. The other outputs are false. This circuit is often used for address decoding.

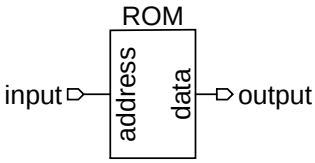


- a *priority encoder* does the inverse operation. The N output bits represent the number of the (highest-numbered) input line.

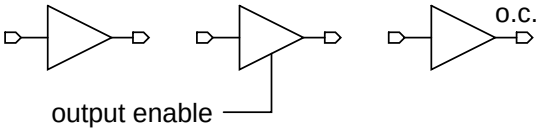
- *adders* and *comparators*, perform arithmetic operations on inputs interpreted as binary numbers



- a *memory* can implement an arbitrary combinational logic function – the input is the address and the stored data is the output.



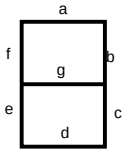
- *drivers* and *buffers* do not alter the logical value but provide higher drive current, tri-state or open drain or open collector (OC) outputs



Exercise 4: Write out the truth table and the canonical (unsimplified) 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:

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

Exercise 5: Fill in the last line of the table. Draw the schematic of a circuit that implements the logic function for the 'g' segment.

VHDL

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

Example 1 - Signal Assignment

Let's start with a simple example – a type of circuit called a `both` 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:

```
-- 'both' : An AND gate

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

architecture rtl of both is
begin
      c <= a and b ;
end rtl ;
```

First some observations on VHDL syntax:

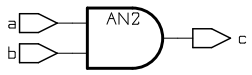
- VHDL is case-insensitive. There are many capitalization styles. I prefer all lower-case. You may use whichever style you wish as long as you are consistent.
- Everything following two dashes "`--`" on a line is a comment and is ignored.
- Statements can be split across any number of lines. A semicolon 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 statements, all of which "execute[2]" at the same time (*concurrently*). This is the critical difference between VHDL and conventional programming languages – VHDL allows us to specify concurrent behaviour.

The single statement in this example 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`. `not` has higher precedence than the other logical operators, all of which have equal precedence. Parentheses can be used to force evaluation in a certain order.

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



Exercise 6: Write a VHDL description for the circuit that would generate the 'a' and 'b' outputs for the 7-segment LED driver shown previously.

Example 2 - Selected Assignment

The selected assignment statement mimics the operation of a multiplexer – the value assigned is selected from several other expressions according to a controlling expression. The following example describes a two-input multiplexer:

```
entity mux2 is
   port (
   a, b : in bit ;
   sel : in bit ;
   y : out bit ) ;
```

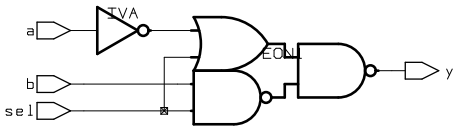
[1] Actually, the V stands for VHSIC. VHSIC stands for Very High Speed IC.

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

```
end mux2 ;

architecture rtl of mux2 is
begin
   with sel select y <=
      a when '0' ,
      b when others ;
end rtl ;
```

which synthesizes to:



Note the following:

- the keyword `others` indicates the default value to assign when none of the other values matches the selection expression. *Always include an* `others` *clause.*
- commas separate the clauses
- the synthesizer assumes active-high logic ('1'≡H)

Example 3 - Bit Vectors

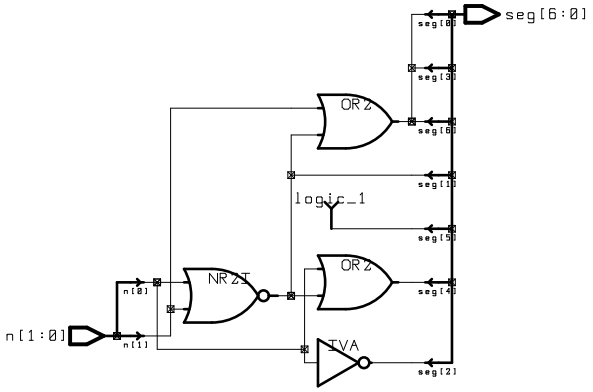
VHDL also allows signals of type `bit_vector` which are one-dimensional arrays of bits that model buses. Using bit_vectors and selected signal assignments we can easily convert a truth table into a VHDL description. The next example is a VHDL description of the 7-segment LED driver:

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

entity led7 is port (
   n: in bit_vector (1 downto 0) ;
   seg: out bit_vector (6 downto 0) ) ;
end led7 ;

architecture rtl of led7 is
begin
   with n select seg <=
      "1111111" when "00" ,
      "0110000" when "01" ,
      "1101101" when "10" ,
      "1111001" when others ;
end rtl ;
```

which synthesizes to:



The indices of bit_vectors can be declared to have increasing (`to`) or decreasing (`downto`) values. `downto` is preferred so that constants read left-to-right in the conventional order.

`bit_vector` constants are formed by enclosing an ordered sequence of binary or hexadecimal values in double quotes after a leading B (optional) or X respectively. For example, `B"1010_0101"` and `X"A5"` are equivalent.

Exercise 7: If `x` is declared as `bit_vector (0 to 3)` and in an architecture 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 ("slices") of vectors can be extracted by specifying a range in the index expression.

Vectors can be concatenated using the '&' operator. For example `y <= x(6 downto 0) & '0'` would set `y` to the 8-bit value of `x` shifted left by 1 bit.

Exercise 8: Write a VHDL description that uses '&' to assign `y` a bit-reversed version of a 4-bit vector `x`.

The logical operators (e.g. `and`) can be applied to `bit_vectors` and operate on a bit-by-bit basis.

Exercise 9: Write a VHDL description for a 2-to-4 decoder using a 2-bit input and a 4-bit output.