Introduction to Logic Design with VHDL

This chapter reviews the design of combinational and sequential logic and introduces the use of VHDL to design combinational logic and state machines.
After this chapter 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,*
- *design a state machine from an informal description of its operation, and*
- *write a VHDL description of a state machine.*

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 ($\overline{B}$), 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 $\overline{OE}$ 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.

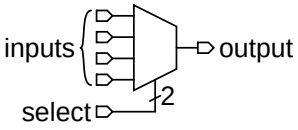
If we form similar terms for the other lines where the desired output variable takes on the value one and then sum all these terms we will have an expression that will evaluate to one only when required and will evaluate to zero in all other cases.

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

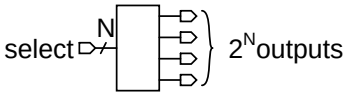
Common Combinational Logic Functions

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.

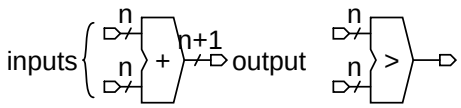


- a *decoder* is a circuit with N inputs and 2^N outputs. The input is treated as a binary number and the output selected by the value of the input is set true. The other outputs are false. This circuit is often used for address decoding.

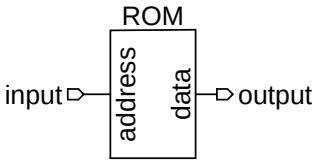


- a *priority encoder* does the inverse operation. The N output bits represent the number of the (highest-numbered) input line.

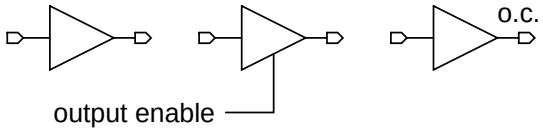
- *adders* and *comparators*, perform arithmetic operations on inputs interpreted as binary numbers



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



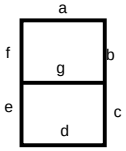
- *drivers* and *buffers* do not alter the logical value but provide higher drive current, tri-state or open drain or open collector (OC) outputs



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:

$a = \overline{A}\,\overline{B} + \overline{A}B + AB$

$b = 1$

$c = \overline{A}\,\overline{B} + A\overline{B} + AB$

$d = \overline{A}\,\overline{B} + \overline{A}B + AB$

$e = \overline{A}\,\overline{B} + \overline{A}B$

$f = \overline{A}\,\overline{B}$

$g =$

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.

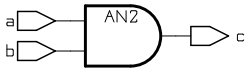
A VHDL description has two parts: an entity part and an architecture part. The entity part defines the input and output signals for the device or "entity" being designed while the architecture part describes the behaviour of the entity.

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 concurrent signal assignment that assigns the value of an expression to the output signal `c`. You can interpret a concurent signal assignment as a physical connection.

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.

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

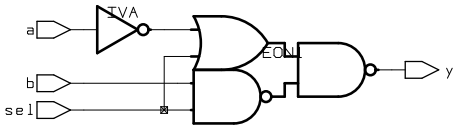
[2] The resulting hardware doesn't actually "execute" but this point of view is useful when using VHDL for simulation.

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 ) ;
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:

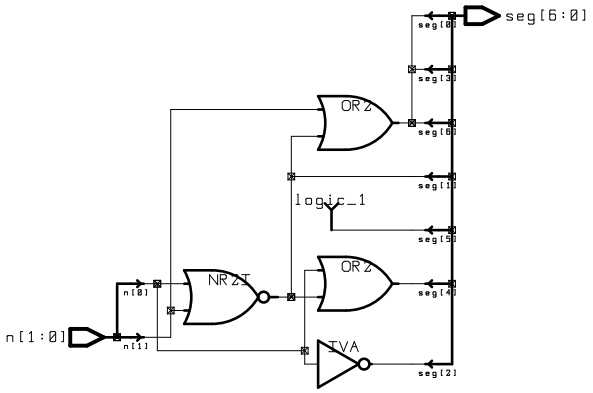
- the keyword `others` indicates the default value to assign when none of the other values matches the selection expression. *Always include an* `others` *clause.*
- commas separate the clauses
- the synthesizer assumes active-high logic ('1'≡H)

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. The range must use the same order ("to" or "downto") as in the declaration.

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`.

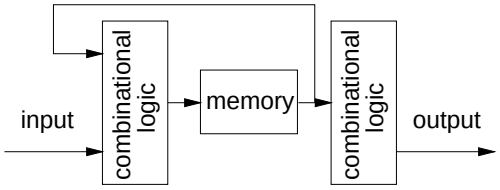
The logical operators (e.g. `and`) can be applied to `bit_vectors` and operate on a bit-by-bit basis.

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

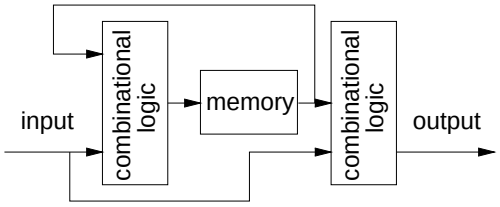
Sequential Logic and State Machines

Sequential logic circuits are circuits whose outputs are a function of their *state* as well as their current inputs. The state of a sequential circuit is the contents of the memory devices in the circuit. All sequential logic circuits have memory.

In theory, *any* sequential logic circuit, even the most complex CPU, can be described as a single state machine (also called a "finite" state machine or FSM). There are two basic types of state machines. In the *Moore* state machine the output is a function only of the current state:



whereas in the *Mealy* state machine the output is a function of the current state and the current inputs:



Moore state machines are simpler and are usually preferred because it's easier to ensure that they will behave correctly for all inputs. However, since their outputs only change on the clock edge they cannot respond as quickly to changes in the input.

Exercise 10: Which signal in the above diagrams indicates the current state?

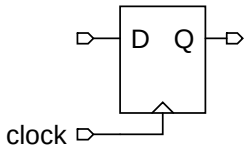
Large sequential circuits such as microprocessors have too much state to be described as a single state machine.

Exercise 11: How may possible states are there for a CPU containing 10,000 flip-flops?

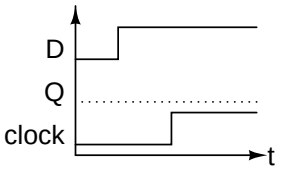
A common approach is to split up the design of complex logic circuits into storage registers and relatively simple state machines. These state machines then control transfers of data between the registers. This type of design is called as *Register Transfer Level* (RTL[3]) design. In this chapter we will study the design of simple FSMs. Later, we will combine these simple state machines with registers to build complex devices.

Common Sequential Logic Circuits

- the *flip-flop* is the basic building block for designing sequential logic circuits. It's purpose is to store one bit of state. There are many types of flip-flops but the only one we will use is the D (delay) flip-flop.



The rising edge of a clock input causes the flip-flop to store the value of the input (typically called "D" and makes it available on the output (typically "Q"). Thus the D flip-flop has a next-state input (D), a state output (Q) and a clock input. The D flip-flop state changes only on the rising clock edge.

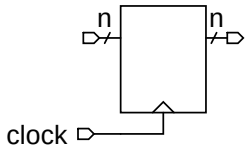


Exercise 12: Fill in the waveform for the Q signal in the diagram above.

Usually all of the flip-flops in a circuit will have the same signal applied to their clock inputs. This *synchronous* operation guarantees that all flip-flops will change their states at the same time and makes it easy to estimate how fast a clock we can use and still have the circuit will operate properly. *Avoid using different clock signals whenever possible!*

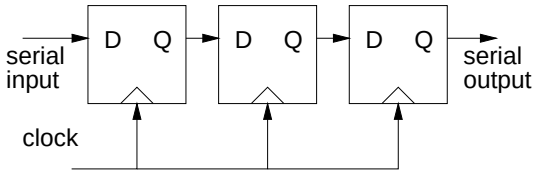
[3]RTL can also mean Register Transfer Language and Register Transfer Logic

- A *register* is several D flip-flops with their clocks tied together so that all the flip-flops are loaded simultaneously.
- A *latch* is a register that whose output follows the input (is *transparent*) when the clock is low.



Exercise 13: What would be another name for a 1-bit register?

- A *shift register* is a circuit of several flip-flops where the output of each flip-flop is connected to the input of the adjacent flip-flop:



On each clock pulse the state of each flip-flop is transfered to the next flip-flop. This allows the data shifted in at one "end" of the register to appear at the other end after a delay equal to the number of stages in the shift register. The flip-flops of a shift registers can often be accessed directly and this type of shift register can be used for converting between serial and parallel bit streams.

Exercise 14: Add the parallel outputs on the above diagram.

- A *counter* is a circuit with an *N*-bit output whose value increases by 1 with each clock. A *synchronous counter* is a conventional state machine and uses a combinational circuit (an adder) to select the next count based on the current count value. A *ripple counter* is a simpler circuit in which the the Q output of one flip-flop drives the clock input of the next counter stage and the flip-flop input is its inverted output.

Exercise 15: Draw block diagrams of two-bit synchronous and ripple counters showing the clock inputs to each flip-flop. Is a ripple counter a synchronous logic circuit?

Design of State Machines

Step 1 - Inputs and Outputs

The first step in the design of a state machine is to specify the the inputs, the states, and the outputs. It's important to ensure these items are identified correctly. If not, the remainder of the design effort will be wasted.

Step 2 - States and Encodings

We then choose enough memory elements (typically flip-flops) to represent all the possible states. *n* flip-flops can be used to represent up to 2^n states (e.g. 3 flip-flops can encode up to 8 states). In some cases we can simplify the design of the state machine by using more than the minimum number of flip-flops.

Exercise 16: If we used 8-bits of state information, how many states could be represented? What if we used 8 bits of state but added the condition that exactly one bit had to be set at any given time (a so-called "one-hot encoding")?

Although it's possible to arbitrarily encode states into combinations of flip-flop values, sometimes a particular encoding of states can simplify the design. For example, we can often simplify the combinational circuit that generates the output by using the outputs as the state variables plus, if necessary, additional flip-flops.

Step 3 - State Transitions

After the inputs, outputs and the state encodings have been determined, the next step is to exhaustively enumerate all the possible combinations of state and input. This is the simplest description of a sequential circuit – a "state transition table" with one line for each possible combination of state and inputs. Then, based on the design's requirements, we determine the required output and next state for each line. In the case of a Moore state machine there will be only one possible output for each state.

Step 4 - Logic Design

Then we design the two blocks of combinational logic that determine the next state and the output. The design of these combinational circuits proceeds as described previously.

Step 5 - Clock and Reset

We also need to apply a clock signal to the clock inputs of the flip-flops. The sequential circuit will change state on every rising edge of this clock signal. Practical circuits will also require some means to initialize (reset) the circuit after power is first applied.

Example: Resetable 2-bit Counter

A resetable 2-bit counter has one input (the reset signal) and two one-bit outputs. A two-bit counter needs four states. Two flip-flops are sufficient to implement four states. The transition conditions are to go from one state (count) to the next state (next higher count) if the reset input is not active, otherwise to the zero state.

In this case we can simplify (eliminate) the design of the output combinational logic by choosing the state variables to be the outputs and the state encodings to be the binary representation of the count.

If we use the variables R as the reset, Q0 and Q1 to represent the state of the system, and Q0' and Q1' as the subsequent state, the tabular representation would be as follows:

Current State		Input	Next State	
Q1	Q0	R	Q1'	Q0'
0	0	0	0	1
0	1	0	1	0
1	0	0	1	1
1	1	0	0	0
X	X	1	0	0

where 'X' is used to represent all possible values (often called a "don't care").

We can obtain the following sum-of-products expressions for these equations:

$$Q1' = \overline{Q1}Q0\overline{R} + Q1\overline{Q0}\,\overline{R}$$
$$Q0' = \overline{Q1}\,\overline{Q0}\,\overline{R} + Q1\overline{Q0}\,\overline{R}$$

Exercise 17: Write the tabular description and draw the schematic of a resetable 2-bit counter with decoded outputs (only one of the four outputs is true at any time). You can assume the counter will always be reset before being used. How does this counter compare to the previous one?

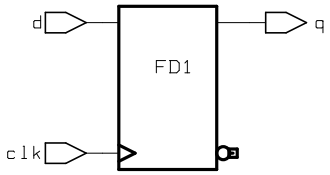
Sequential Circuits in VHDL

The design of sequential circuits in VHDL requires the use of the `process` statement. Process statements can include *sequential* statements that execute one after another as in conventional programming languages. However, for the logic synthesizer to be able to convert a process to a logic circuit, the process must have a very specific structure. You should use only the simple three-line type of process shown below. This is because signal assignments within a process do not happen as you might expect and may lead to strange results. **In this course you may only use processes to generate registers and may only use the single-if structure shown below.**

As an example, we can describe a D flip-flop in VHDL as follows:

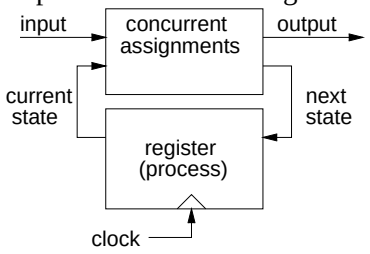
```
entity d_ff is port (
   clk, d : in bit ;
   q : out bit ) ;
end d_ff ;

architecture rtl of d_ff is
begin
   process(clk)
   begin
      if clk'event and clk = '1' then
         q <= d ;
      end if ;
   end process ;
end rtl ;
```

The expression `clk'event` (pronounced "clock tick event") is true when the value of `clk` has changed since the last time the process was executed. This is how we model "memory" in VHDL. In the process the output `q` is only assigned a value if `clk` changes and the new value is 1. When `clk` = 0 the output retains its previous value. It's necessary to check for `clk`=1 to distinguish between rising and falling edges of the clock.



FSMs in VHDL are implemented by using concurrent assignment statements (e.g. selected assignments) to generate (1) the output and (2) the next state from (a) the current state and (b) the inputs. The process shown above is used to generate the flip-flops that define the current state.

This corresponds to the following block diagram:



A VHDL description for a 2-bit counter could be written as follows:

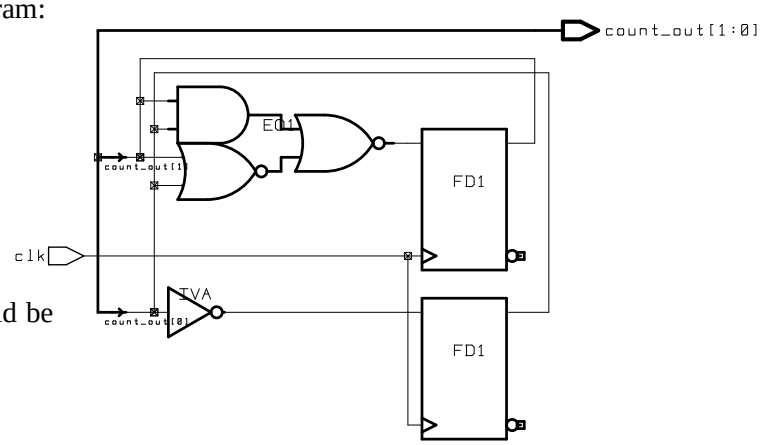
```
entity count2 is port (
   clk : in bit ;
   count_out : out bit_vector (1 downto 0) ) ;
end count2 ;

architecture rtl of count2 is
   signal count, next_count : bit_vector (1 downto 0) ;
begin
   -- combinational logic for next state
   with count select next_count <=
      "01" when "00",
      "10" when "01",
      "11" when "10",
      "00" when others ;

   -- combinational logic for output
   count_out <= count ;

   -- sequential logic
   process(clk)
   begin
      if clk'event and clk = '1' then
         count <= next_count ;
      end if ;
   end process ;
end rtl ;
```

Exercise 18: Draw the block diagram corresponding to this VHDL description. Use a multiplexer for the selected assignment and and a register for the state variables. Label the connections, inputs and outputs with the corresponding VHDL signal names.

Note that two signals need to be defined for each flip-flop or register in a design: one signal for the input (`next_count` in this case) and one signal for the output (`count`).

Also note that we must define a new signal for an output if its value is to be used within the architecture. This is simply a quirk of VHDL.

The synthesized circuit is:



Exercise 19: Identify the components in the schematic that were created ("*instantiated*") the different parts of the VHDL code.

Exercise 20: Modify the diagram of the 2-bit counter by adding a (synchronous) reset input and a 2-input, 2-bit multiplexer. Draw a block diagram showing the modified circuit. Label the connections and i/o with appropriate signal names. Write the corresponding VHDL description.