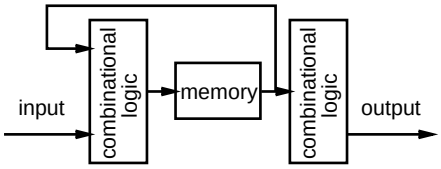
Sequential Logic Design

This lecture reviews the design of sequential logic and shows how state machines can be described in VHDL. After this lecture you should be able to: (1) design a state machine from an informal description of its operation, and (2) write a VHDL description of the state machine.

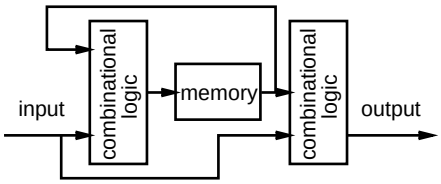
Sequential Logic and State Machines

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

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



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



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

However, large sequential circuits such as microprocessors have too much state to be easily described as a single state machine. A common approach is to split up the design into storage registers and a state machine that controls transfers between the registers. This is known as *Register Transfer Level* design. In this lecture we will study the design of simple FSMs.

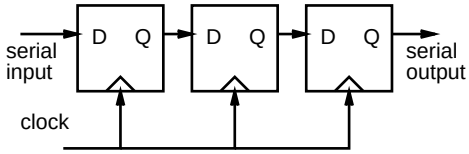
Common Sequential Logic Circuits

A *flip-flop* is the simplest sequential logic circuit. It's purpose is to store one bit of state. There are many types of flip-flops but by far the most common is the D (delay) flip-flop. The rising edge of a clock stores 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 clock edge. Usually all of the flip-flops in a circuit will have the same signal applied to their clock inputs. This *synchronous* operation guarantees that their states will change at the same time.

A *latch*[1] or *register* is several D flip-flops with their clocks tied together so that all the flip-flops are loaded simultaneously.

Exercise: 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: 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

[1] Strictly speaking a latch is a register that whose output follows the input (is *transparent*) when the clock is low.

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. This is an example of a non-synchronous design because the flip-flops have different clocks.

Design of State Machines

The first step in the design of a state machine is to specify the the inputs, the states, the outputs, and the conditions required to change states.

We must 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: 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")?

As with combinational logic, the simplest description of a sequential circuit is as a table with one line for each possible combination of state and inputs. After the inputs and the number of state variables has been determined, the next step is to exhaustively enumerate all the possible combinations of state and input. Then, based on the design's requirements, we determine the required output and next state for each line.

The final step is to design the block of combinational logic that determines the next state and the output. This combinational circuit has two sets of inputs: the outputs of the flip-flops (representing the current state) and the input to the sequential circuit. The circuit also has two sets of outputs: one is connected to the inputs of the flip-flops (the next state) and other to the output of the sequential circuit. The designs of these combinational circuits proceeds as described in the section on combination circuits.

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 when power is first applied.

Example: Synchronous 2-bit Counter

A two-bit counter will have four states. Two flip-flops are sufficient to implement four states. In this case there are no inputs, the circuit merely counts up at each clock signal. The transition conditions are simply to unconditionally go from one state (count) to the next state (next higher count).

If we use the variables 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:

Q1	Q0	Q1'	Q0'
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

This example is particularly simple since there are no inputs and the outputs are the same as the values of the state variables. The combinational circuit only needs to determine the next state based on the current state. We can obtain the following sum-of-products expressions for these equations:

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

Exercise: Write the tabular description of a counter with an up/$\overline{\text{down}}$ input that controls the count direction.

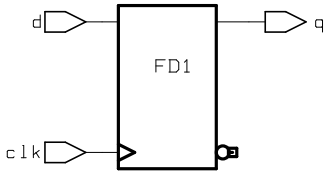
Sequential Circuits in VHDL

We have seen that if an output signal is always assigned a new value each time the process "executes" then the output is only a function of the inputs, no memory is required, and a combinational circuit will be synthesized. However, if in some cases an output signal would not be assigned a new value then it needs to retain its previous value, memory is required to ensure proper operation, and a sequential logic circuit will be synthesized. This is how sequential circuits are synthesized from VHDL descriptions.

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

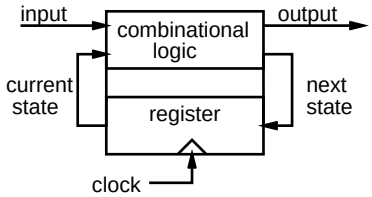
```
entity D_FF is
   port ( clk, d : in bit ;
   q : out bit ) ;
end D_FF ;

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. In this case 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.



One way of representing a FSM in VHDL is to use two processes. One process implements the combinational logic and computes the output and next state based on the current state and inputs. The second process is sensitive to the clock and sets the current state equal to the value for the next state as computed by the first process.

This corresponds to the following block diagram:

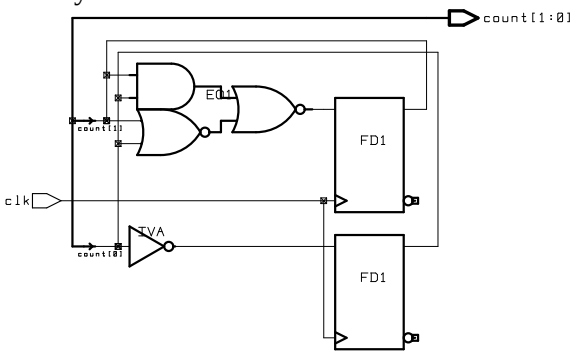


For example, a VHDL description for a 2-bit counter could be written as follows:

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

architecture rtl of count2 is
   signal current, nexts :
      bit_vector (1 downto 0) ;
begin
   -- combinational logic:
   process(current)
   begin
     -- compute next state and output
     -- from current state and input
     case current is
        when "00" =>
           nexts <= "01" ;
           count <= "00" ;
        when "01" =>
           nexts <= "10" ;
           count <= "01" ;
        when "10" =>
           nexts <= "11" ;
           count <= "10" ;
        when "11" =>
           nexts <= "00" ;
           count <= "11" ;
        end case ;
   end process ;

   -- sequential logic
   process(clk)
   begin
      if clk'event and clk = '1' then
         current <= nexts ;
      end if ;
   end process ;

end rtl ;
```

Exercise: Modify the above description to add an up/down control input.

The synthesized circuit is:



Exercise: Identify the components in the schematic that were created ("*instantiated*") by each process.

Signal Assignment in Processes

Should you be tempted to assign a value to a signal more than once in a process you should note that signal assignments do not take effect until the *end* of the process[2]. Consider the case where the following two

[2] If you want to make use of variables with conventional behaviour within a process you need to use VHDL `variable`s (to be described later)

statements appear in a process:

```
z <= '1' ;
z <= not z ;
```

Exercise: What is the value of z? 1? 0? something else?

Simulation of VHDL Descriptions

Although it's possible to check the simple designs we've built so far by looking at the resulting schematic, this is difficult for complex circuits. Therefore we normally simulate the behaviour of VHDL descriptions before implementing them in order to make sure the resulting circuit will behave correctly.

To test behaviour through simulation we write a VHDL entity called a *test bench* whose purpose is to exercise a design. The test bench applies sequences of known values to the inputs of the entity being tested and checks the entity's outputs to ensure the design behaves as expected. The input/output test signals are known as *test vectors*.

In this course you will be supplied with test benches for each assignment requiring simulation.