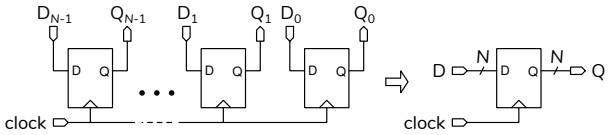
Sequential Logic and State Machines

This lecture describes how registers, counters, shift registers, timers, and arbitrary state machines are implemented in Verilog and how they are documented.
You should be able to design and document state machines using: an informal description, truth tables, state transition diagrams, and synthesizable Verilog. You should be able to convert between any of these descriptions. You should be able to select an appropriate state encoding.

Introduction

Registers

We can connect N flip-flops to the same clock to form an N-bit *register*. This common clock loads every flip-flop at the same time:



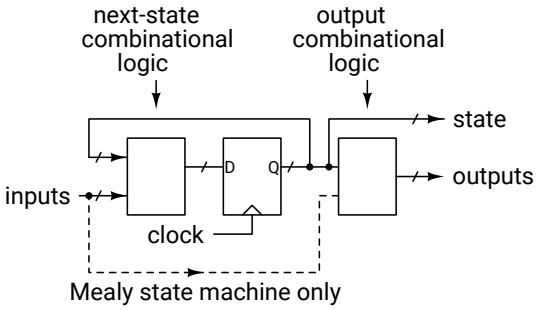
The Verilog **`always_ff`** statement creates a register.

Modern logic design uses the same periodic clock signal for every register. The state register(s) are thus loaded on each rising edge of the clock, but not necessarily with a different value.

State Machines

The *state* of a register is its value. A *state machine* is a description of how the state changes.

A state machine is implemented as a register whose value on the next rising edge of the clock is based on the current state and inputs:



The state register is loaded on each rising edge of the clock, but if it's loaded with its current value then there is no change of state.

The output of a state machine is a function of its state. In some cases the state itself may be the output.

The above describes a *Moore* state machine. A *Mealy* state machine is one where the output is a function of the current state as well as the inputs. We will only consider Moore state machines.

Counters

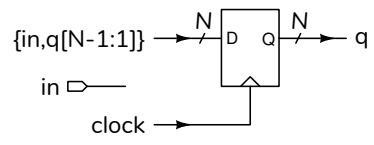
A *counter* is a common state machine. The register's value increases by one on each clock edge. Typical inputs include those to restart the sequence (typically called reset), to pause or continue the sequence (hold or enable), or to count down instead of up (e.g. up/-down).

Exercise 1: Write the Verilog for an 8-bit counter named '**n**'. Modify it so that when the register value is set to 0 each time it reaches 99. Draw the corresponding block diagram. Identify the next-state combinational logic.

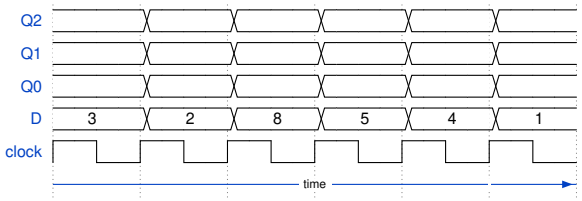
Exercise 2: Write a 4-bit counter named '**`cnt`**' with **`reset`**, **`enable`**, and **down** inputs. **`reset`** should have priority over **`enable`**. Draw the block diagram.

Shift Registers

A *shift register* is a register whose next value (state) is an input bit concatenated with the current register bits as shown in the following diagram:



Exercise 3: The example above is an N-bit shift register that shifts the bits right. Draw a block diagram and write the Verilog for a 6-bit shift register that shifts left.



Exercise 4: Fill in the diagram above for a 4-bit ($N = 4$) right-shift shift register. Assume the initial value is zero. Which bit is the oldest (first) value in the waveform? Which bit of the shift register holds the oldest value?

A shift register makes previous inputs available in parallel. This is useful for detecting sequences in an input.

Exercise 5: Draw a block diagram and write the Verilog for a circuit that sets an output named **`detect`** high when the sequence of values 1, 1, 0, 1 has appeared on an input named **`in`** on successive rising edges of the clock.

Timers and Clock Dividers

A common application for counters is to implement delays or periodic signals.

It takes N clock periods for a counter to count down[1] from $N-1$ to 0. If the clock period is T seconds then the time taken is NT seconds. A circuit can thus create a delay of NT seconds by counting N clock cycles[2]

Exercise 6: What value of N would result in a 20 ms delay if the clock frequency is 50 MHz? How many bits are needed for this timer's register?

If the counter is reset to $N-1$ when it reaches 0 then the count values will be periodic with a period NT. If some event happens each time the count reaches a specific value (e.g. 0) then this event happens with period NT (and thus a frequency $1/NT$).

Exercise 7: Assume the timer above is reset to $N-1$ each time it reaches 0. What is the time between each time the count reaches zero? What is this frequency? For how long does the register have the value 0? What are the period and frequency of a signal that is inverted each time the count reaches 0?

[1] Timers traditionally count down from $N-1$ to 0 rather than up from 0 to $N-1$ because it's simple to determine when the count reaches 0: adding -1 does not cause a carry.

[2] The time includes clock cycles during which the counter has values $N-1$ through 0 inclusive.

State Machine Descriptions

Consider a state machine with two bits of state that sequences through the values 00, 01, 10, 11 and back to 00. The output should be 0 in states 00 and 01 and 1 in states 10 and 11. There is an input named reset. The state is set to 00 if reset is 1.

Truth Tables

We can describe a state machine using truth-tables. One table has columns for the current state, the input value(s), and the corresponding next state. Another table has columns for the current state and the output.

We could write the state transition table and the output table as:

state	reset	next state
00	0	01
01	0	10
10	0	11
11	0	00
00	1	00
01	1	00
10	1	00
11	1	00

state	output
00	0
01	0
10	1
11	1

Exercise 8: For the state machine described above, if the current state is 01, what will be the next state? When will the state change? What is the output in state 00? In state 01? In state 10?

Simplifications

We can simplify truth tables using the following conventions: (1) x can be used for "don't care" in state or input columns; (2) the first matching row applies; (3) if there is no matching row there is no change of state; (4) expressions can be used to define the next state as a function of the current state and the input.

For example, we could rewrite the above state transition table as:

state	reset	next state
xx	1	00
11	0	00
n	0	$n+1$

Exercise 9: What will be the next state if the state is 00 and the `reset` input is 1? If the state is 00 and the `reset` input is 0? When does the state change? When does `reset` affect the output?

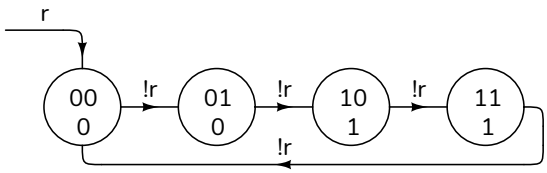
States can be labelled with names instead of numerical values.

Exercise 10: Write the above state transition and output tables using state names A, B, C, and D.

State Transition Diagrams

A state machine can also be described by a state transition diagram drawn using the following conventions: (1) each state is represented by a circle labelled with the state name or value; (2) each state shows the output for that state (unless the state is also the output value); (3) arrows show possible transitions between states; (4) transitions are labelled with an expression that must be true (non-zero) for the transition to happen[3]; (5) unlabelled transitions happen unconditionally; (6) transitions with no origin come from every state; (7) conditions that don't cause a change of state are not shown[4]; (8) only one transition out of a state may be true (the expressions on the arrows leaving each state must be mutually exclusive).

The following is a state transition diagram for the state machine above:



r: reset==1

Exercise 11: Modify the diagram so the state machine counts to 11 and stops.

Exercise 12: Add a **down** input that cause the values to count down.

State Machines in Verilog

A state machine can be written in Verilog using one conditional operator for each row in the state transition table or for each transition in the state transition diagram. The condition in the expression is expression that labels the arrow in the diagram or an expression that is true for the current state and inputs for that transition. The true value is the next state for that transition. The false value is the next conditional operator or, if no transition matches, the current state.

A straightforward translation of the truth tables above would be written as:

```
// 2-bit clock divider with reset

module ex79
   ( output logic [1:0] count,
     output logic out,
     input logic reset, clk ) ;

   always_ff @(posedge clk) count
      <=  reset && count == 2'b00 ? 2'b00 :
          reset && count == 2'b01 ? 2'b00 :
          reset && count == 2'b10 ? 2'b00 :
          reset && count == 2'b11 ? 2'b00 :
         !reset && count == 2'b00 ? 2'b01 :
         !reset && count == 2'b01 ? 2'b10 :
         !reset && count == 2'b10 ? 2'b11 :
         !reset && count == 2'b11 ? 2'b00 :
         count ;

   assign out
     = count == 2'b00 ? 1'b0 :
       count == 2'b01 ? 1'b0 :
       count == 2'b10 ? 1'b1 : 1'b1 ;

endmodule
```

— or more simply as —

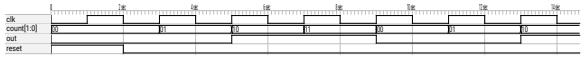
```
// 2-bit clock divider with reset

module ex67
   ( output logic [1:0] count,
     output logic out,
     input logic reset, clk ) ;

   always_ff @(posedge clk) count
      <= reset ? 2'b00 :
         count == 2'b11 ? 2'b00 :
         count + 1'b1 ;

   assign out = count == 2'b10 || count == 2'b11 ;

endmodule
```



State Encodings

States may be represented ("encoded") in different ways:

Binary States are encoded as binary numbers. This requires the fewest number of flip-flops. It is

[3] Abbreviations for these expressions can be used to keep the diagram tidy.

[4] Some authors do not allow this.

used for state machines with many states such as counters. In the above example the binary encoding of the states would be `00`, `01`, `10` and `11`.

One-Hot There is one flip-flop for each state. Only one flip-flop at a time can be set to 1. This is used when there are few states because simplifies the next-state and output logic. In the above example a one-hot encoding might be `1000`, `0100`, `0010` and `0001`.

Output Each state is encoded as the output for that state. This eliminates the need for any logic to determine the output as a function of the state. However, this is only possible when the output is different for each state. The above example could not use an output state encoding because the output is `0` for states `00` and `01` and the output is `1` for states `10` and `11`.

Input The state encoding is a sequence of previous inputs. This is used when the output can be determined from a small number of previous inputs. For example, a combination lock might unlock after the correct sequence of inputs.

Exercise 13: How many bits need to be considered to detect a specific state when a binary encoding is used? How many need to be considered if a one-hot encoding is used?

Exercise 14: If we used 8-bits of state information, how many states could be represented? What if we used 8 bits of state but used a "one-hot" encoding?