More Verilog

This lecture describes the relationship between numbers, logic levels and truth values. It also describes Verilog modules and parameters.
After this lecture you should be able to: convert between high/low logic levels and true/false truth values for active-high and active-low interfaces, declare modules with parameters and ports, and instantiate modules using positional, named and wildcard parameters and signals.

Numbers, Truth Values and Logic Levels

Numbers are used for counting, logic levels are voltages, and truth values can be true or false. These are different, but related.

0 and 1 almost always mean false and true respectively. But there are two common conventions for logic levels and truth values. Active-high signals are true when they are high and active-low signals are true when they are low.

Active-low signals can be denoted by:

- a bar over the signal name ($\overline{\text{reset}}$)
- an asterisk after the signal name (`RESET*`)
- a suffix of N (or n) after the signal name (`reset_n`)

Verilog uses the usual meaning of 0 and 1 as truth values in expressions. But for inputs and outputs it always uses `0` and `1` for low and high respectively. Thus the truth value of `0` or `1` in Verilog depends on the context – when used for I/O a `0` could mean either true or false.

The following table summarizes the correspondence between active-high and active-low signals, truth values, logic levels and the values used in Verilog:

signal name	logic level	truth value	Verilog expression	Verilog I/O
$\overline{\texttt{s}}$	L	T	1	0
$\overline{\texttt{s}}$	H	F	0	1
s	L	F	0	0
s	H	T	1	1

Exercise 1: Is a signal named $\overline{\textbf{overload}}$ active-high or active-low? Is there an overload if this signal is high? What if the signal was named **overload**?

Exercise 2: Come up with active-high and an active-low names for a signal that is at 3 V when a door is open and 0 V when the door is closed.

In addition to the two ways to represent truth values with voltages (active-high and active-low) there are also two ways to represent binary digits (“bits”) with voltages. A high voltage may represent either a 0 or a 1. Signals where a 1 is represented by a low voltage typically, but not always, use active-low notation.

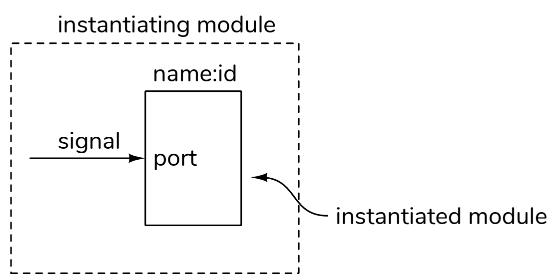
Exercise 3: If $\overline{\texttt{D}}$ is a word and $\overline{\texttt{D[0]}}$ is low, is the word an even or odd number?

Modules

Simple things are easier to design and test than complex ones. Thus it’s good practice to divide designs into smaller parts[1]. These can often be re-used.

Many designs incorporate complex parts designed by others (e.g. processors, memories and interfaces), called design IP (“Intellectual Property”).

In Verilog each part is a **`module`**. Modules describe logic that can be “instantiated” (duplicated and inserted into) another module:



[1]How small? A good rule of thumb is to make sure each part can be described on a single page.

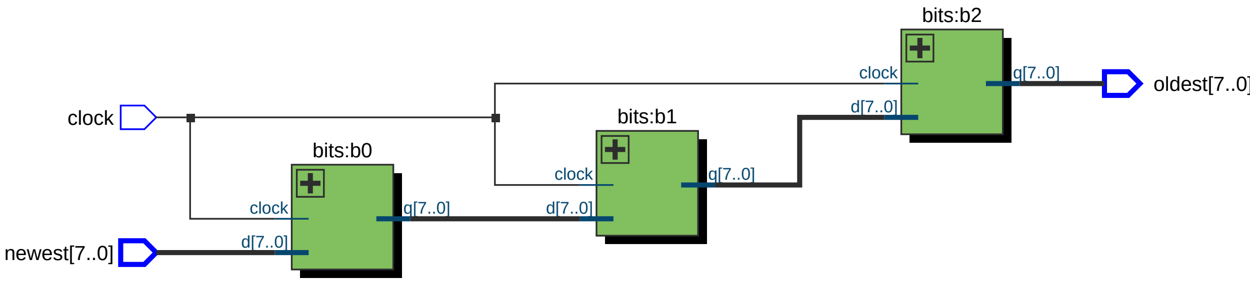


Figure 1: Shift Register Synthesis

The module's interfaces are defined by a header describing ports and parameters. Ports are **in**, **out** or **inout** (bidirectional) signals while parameters are values that can customize each instance of a module. The module's body contains additional signal declarations and parallel (concurrently executing) statements between **module** and **endmodule**. These define the structure or behaviour of the module.

Here's an example of a module named **bits** that defines an **nb**-bit register:

```
module bits
  #(parameter nb=1)
   (
    input  logic [nb-1:0] d,
    output logic [nb-1:0] q,
    input  logic clock
    ) ;

   always_ff @(posedge clock) q <= d ;

endmodule
```

The parameter **nb** has a default value of **1** which is used if a value is not specified when this module is instantiated. There are two input ports (named **d** and **clock**) and one output port (named **q**).

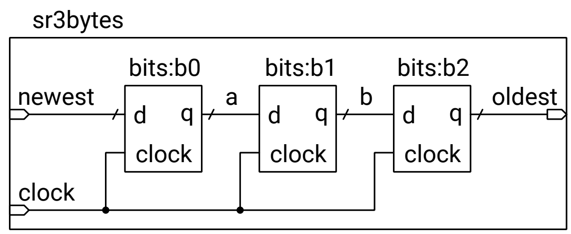
A module instantiation starts with the name of the module followed by parameter values (if any), an instance name (to identify individual instances of the same module), and a description of how to connect signals in the instantiating module to the ports in the instantiated module. For example:

```
bits #(4) b0 (a,b,c) ;
```

Would instantiate a **bits** module with one parameter of value 4, an instance name **b0** and connect the signals **a**, **b** and **c** in the instantiating module to the corresponding ports in an instance of the **bits** module (**d**, **q** and **clock** respectively).

Exercise 4: Draw a diagram for this instantiation of the **bits** module. Label the module, instance, signal and port names as in the diagram above.

An 8-bit, 3-stage shift register could be built using three **bits** modules:




```
module sr3bytes
  (
   input  logic [7:0] newest,
   output logic [7:0] oldest,
   input  logic clock
   ) ;

  localparam nbits = 8 ;

  logic [nbits-1:0] a, b ;

  // matching by order
  bits #(nbits) b0 (newest,a,clock);

  // matching by name (order does not matter)
  bits #(.nb(nbits)) b1 (.q(b),.clock,.d(a));

  // wildcards for names that match
  bits #(.nb(nbits)) b2 (.d(b),.q(oldest),.*);

endmodule
```

Exercise 5: Identify the module instantiation statements in the code above. For each one, what is the instantiated module's name? The instance name?

When one module is instantiated in another, a signal can be connected to module port by:

- port order (**signal**),
- port name and explicit signal name (**.port(signal)**),
- port name only – connecting to the matching signal name (**.port**),

- a wildcard that matches all remaining matching port and signal names (`.*`).

The signal name can be an expressions (e.g. `word[15:8]`) instead of a signal. Matching of values to parameters can be done by order (`value`) or explicitly, `.parameter(value)`.

The synthesis result, shown above, is as expected.