Hierarchical Design

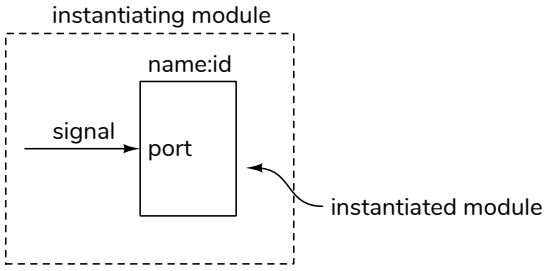
This lecture describes Verilog modules and parameters.
After this lecture you should be able to: declare modules with parameters and ports, and instantiate modules using positional, named and wildcard parameters and signals.

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:



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
```

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

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`**).

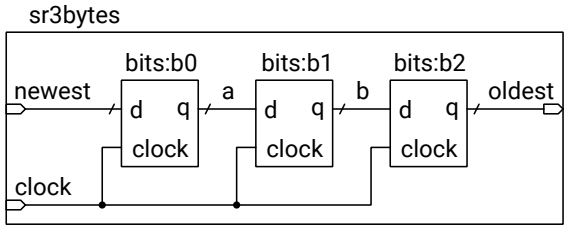
A module instantiation starts with the name of the module followed by parameter values (if any), an instance name (to identify individual instances of the same module), and a description of how to connect signals in the instantiating module to the ports in the instantiated module. For example, the statement:

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

would instantiate a **`bits`** module giving the first parameter a value 4, giving this instance the name **`b0`**, and connect the signals **`a`**, **`b`** and **`c`** in the instantiating module to the corresponding ports in a copy (an instantiation) of the **`bits`** module (in this case, **`d`**, **`q`** and **`clock`** respectively).

Exercise 1: 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)
```

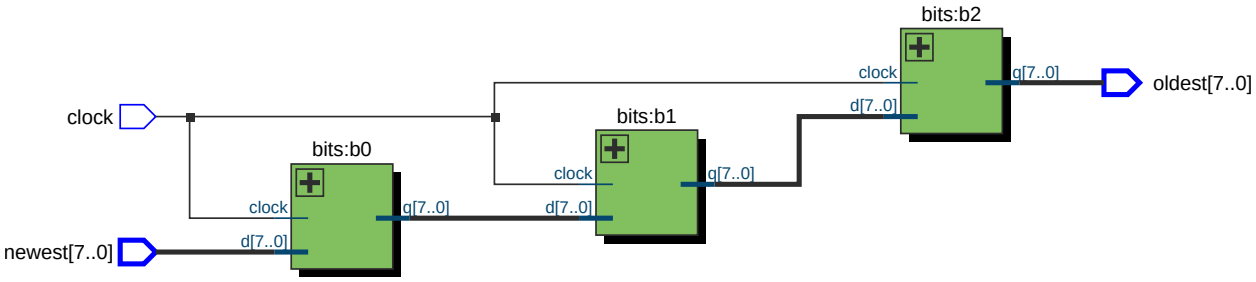


Figure 1: Shift Register Synthesis

```
bits #(.nb(nbits)) b1 (.q(b),.clock,.d(a));

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

endmodule
```

Exercise 2: 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.