ELEX 2117 : Digital Techniques 2
2022 Winter Term

More Verilog

This lecture describes Verilog modules and parameters, arrays and the relationship between numbers, logic levels and truth values.
After this lecture you should be able to:

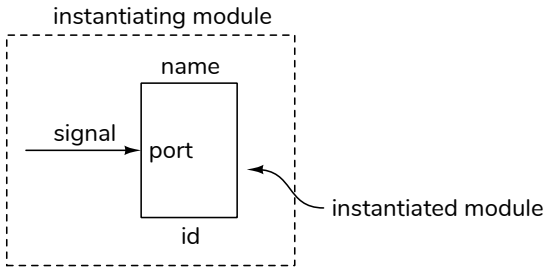
- *declare modules with parameters and ports*
- *instantiate modules using positional, named and wildcard parameters and signals*
- *declare arrays of arrays and use initialized arrays as lookup tables*
- *convert between high/low logic levels and true/false truth values for active-high and active-low interfaces*

Modules

It's good practice to divide designs into smaller parts[1] because small, simple circuits are easier to design and test than large ones.

With careful partitioning there can be other advantages. A larger part could use several copies of the same part. Or a part might be re-used in future designs.

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.

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

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
    ) ;

   logic [nb-1:0] q_next ;

   assign q_next = d ;

   always_ff @(posedge clock) q = q_next ;

endmodule
```

The parameter **`nb`** has a default value of **`1`** which is used if a value is not given when the 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 b0 (a,b,c) ;
```

Would instantiate a **`bits`** module with instance name **`b0`** that connected the signals **`a`**, **`b`** and **`c`** to the corresponding ports in the **`bits`** module (**`d`**, **`q`** and **`clock`** respectively).

Exercise 1: Draw a diagram labelling 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:

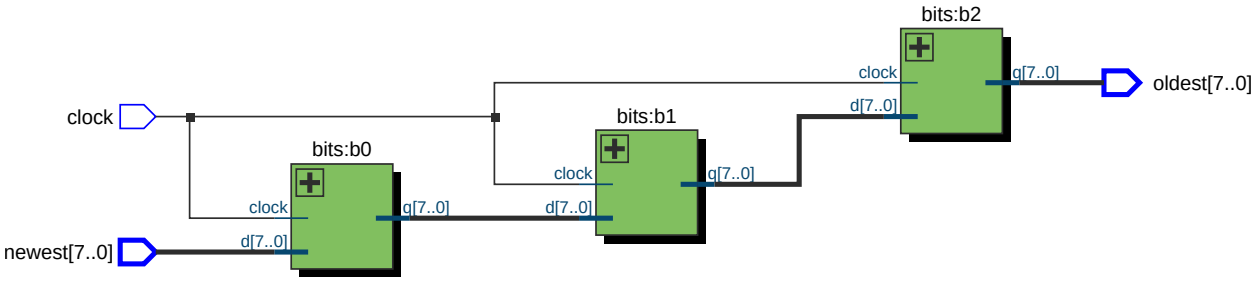
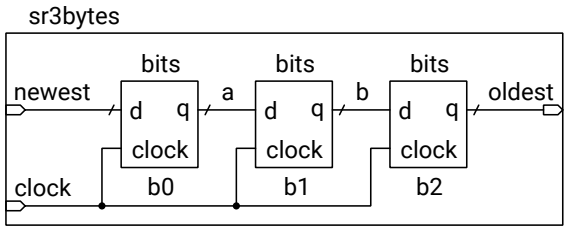
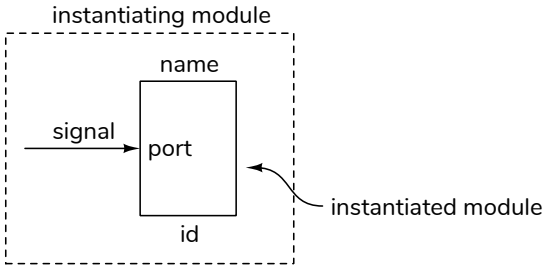


Figure 1: Shift Register Synthesis




```
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 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:

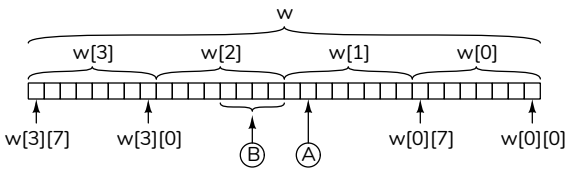
- port order (`signal`),
- port name and explicit signal name (`.port(signal)`),
- port name only – connecting to the matching signal name (`.port`),
- wildcard that matches all remaining matching port and signal names (`.*`).

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.

Arrays of Arrays

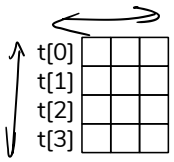
We can also declare arrays of arrays. For example, `logic[3:0][7:0] w;` declares `w` as an array of four arrays of eight bits. A single index selects one of the bytes, two indices selects a byte and a bit from that byte:



Exercise 3: How would you specify the bit labelled A in the diagram above? The bits labelled B?

Lookup Tables

Initialized arrays can serve as "lookup tables" – tables where the index value selects a row. For example, a lookup table with four three-bit values would be declared as `logic [0:3][2:0] t ;` and drawn as:



Note that the rows indices are labelled in the conventional order – increasing from top to bottom.

Lookup tables can implement any function by initializing the rows with the function value for the corresponding row index.

For example, the **onebits** lookup table in the following code contains the number of '1' bits in a binary number:

```
                        // 0 1 2 3 4 5 6 7
logic [0:7][2:0] onebits = '{0,1,1,2,1,2,2,3} ;
logic [3:0] n ;
...
assign ... = onebits[n] ;
```

and the value of the expression **onebits[n]** is the number of '1' bits in **n**.

Exercise 4: Write a Verilog lookup table to look up whether a value between 0 and 7 is a prime number or not. The result should be 1 if the value is a prime or else 0. *Hint: The primes are 2, 3, 5 and 7.*

Unpacked Arrays

Arrays may also have "unpacked" dimensions that appear after the signal name and model memories. For example: **logic [7:0] rom [32] ;** would model a 32-byte memory with array indices 0 to 31.

In array references, the unpacked dimension(s) (if any) are specified first, followed by the packed dimensions (if any). For example, **rom[31][0]** would be the least-significant bit of the last word in the **rom** above.

Numbers, Logic Levels and Truth Values

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

Almost universally, 1 and 0 are synonymous with true and false respectively. But there are two common conventions for the relationship between logic levels and truth values:

Number	Truth	Active High (Verilog)	Active Low
0	F	L	H
1	T	H	L

Verilog uses the active-high convention for inputs and outputs: **0** and **1** are treated as low and high respectively.

But whether **1** or **0** indicates that a signal is true or false depends on the context. In a logical expression, **1** would mean that the signal was true. But as an input or output level **1** could mean that the signal was true (for active-high signals) or false (for active-low signals).

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**)

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 5: 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 6: 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.

Exercise 7: If $\overline{\text{D}}$ is a data bus and $\overline{\text{D0}}$ is low, is the value on the data bus an even or odd number?

A Bit More Syntax

The notations **'0** and **'1** are convenient abbreviations for a literal constant that is all-zeros or all-ones.