

Introduction to Digital Design with Verilog HDL

*This is a brief introduction to digital circuit design using the System Verilog Hardware Description Language (Verilog HDL). After this lecture you should be able to: define a module with single- and multi-bit **logic** inputs and outputs; write Verilog numeric literals in binary, decimal and hexadecimal bases; declare arrays and arrays of arrays; evaluate the value and width of expressions containing **logic** signals, arrays, numeric literals and the operators described below; use **assign**, **always_ff**, and component instantiation statements to create combinational logic, registers, and to instantiate one module in another.*

Introduction

Hardware Description Languages (HDLs) are used to design digital circuits. In this course we will use System Verilog, the modern version of the Verilog HDL, rather than the other popular HDL, VHDL.

Let's start with a simple example – a circuit called an **ex1** that has one output (**y**) that is the logical AND of two input signals (**a** and **b**). The file **ex1.v** contains the following Verilog description:

```
// AND gate in Verilog

module ex1 ( input logic a, b,
             output logic y );

    assign y = a & b ;

endmodule
```

Some observations on Verilog syntax:

- Everything following `//` on a line is a comment and is ignored.
- Module and signal names can contain letters, digits, underscores (`_`), and dollar signs (`$`). The first character of an identifier must be a letter or an underscore. They cannot be the same as certain reserved words (e.g. **module**).
- Verilog is case-sensitive: **a** and **A** would be different signals.
- Statements can be divided into any number of lines. Any number of spaces can be used. A semicolon ends each statement.

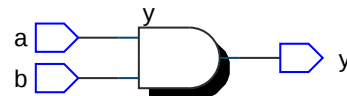
Capitalisation and indentation styles vary. In this course you will need to follow the coding style guide available on the course website.

The module definition begins by defining the input and output signals for the device being designed.

The body of the module contains one or more statements, each of which operates at the same time – *concurrently*. This is the key difference between HDLs and programming languages – HDLs allows us to define concurrent behaviour.

The single statement in this example is a signal assignment that assigns the value of an expression to the output signal **y**. Expressions involving **logic** signals can use the logical operators described below including **&** (and), **|** (or), and **^** (exclusive-or). Parentheses can be used to order the operations.

From this Verilog description a program called a logic synthesizer (e.g. Intel's Quartus) can generate a circuit that has the required functionality. In this case it's not too surprising that the result is the following circuit:



If you're familiar with the C programming language you'll note that Verilog uses similar syntax.

Exercise 1: What changes would result in a 3-input OR gate?

Exercise 2: What schematic would you expect if the statement was `assign y = ( a ^ b ) | c ;`?

Reserved Words

System Verilog has about 250 reserved words (including many common ones such as **time**, **wait**, **disable**, **reg**, **table**, **input**, ...) that may not be used as module or signal names. Doing so will give a syntax error when you try to compile the description. An editor with syntax highlighting will help you identify and avoid using reserved words.

Variables and Literals

Logic Signals

Verilog's **logic** variables – also called “signals” – can have four values: **0** (false or low), **1** (true or high), **z** (high impedance, used for synthesis of tri-state outputs) and **x** (undefined, represents an undefined value).

Verilog's other types (**signed**, **int**, **float**, **char**, **struct**, ...) are not needed for this course.

Packed Arrays

An array is a collection of **logic** signals whose elements can be selected by a value called the index. Arrays often represent numerical values in binary form.

For example, the declaration **logic [3:0] a;** specifies an array named **a** with a ‘width’ of four bits.

In this example the bit indices are declared as going from 3 down to 0. If the bit values are written out and represent a binary number then **a[3]**, the left-most bit, is the most significant bit. **a[0]** is the right-most (least significant) bit.

Exercise 3: If the signal **i** is declared as **logic [2:0] i;**, what is the ‘width’ of **i**? If **i** has the value 6 (decimal), what is the value of **i[2]**? Of **i[0]**?

Packed arrays synthesize to buses or words.

Numeric Literals

Numeric literals, often called “constants,” are written as a sequence of up to three parts: the number of bits; the letters ‘**b**’ for binary, ‘**h**’ for hex, or ‘**d**’ for decimal; and the value in the specified base. The default width is 32 and the default base is decimal. Underscores (**_**) may be used within the value to improve readability.

Exercise 4: What are the widths and values, in decimal, of the following: **4'b1001**? **5'd3**? **6'h0_a**? **3**?

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

Unpacked Arrays

An unpacked array is a collection of packed arrays that can be accessed one at a time. The unpacked dimensions are given after the name and are often in increasing order. For example, **logic [7:0] a [0:15];** might describe a memory with 16 words of 8 bits.

Array Literals

Array literals can be defined by grouping with ‘**{...}**’. For example,

logic [3:0] z [0:1] = '{ 3'b11, 3'b101 };

Exercise 5: Draw a two-dimensional table showing the values of the bits in **z**.

Array literals define truth tables. Each element of the array corresponds to one row of the table. The array index is the input that selects the output word.

Exercise 6: Write the truth table for a one-bit adder with carry. There are three inputs (**a**, **b** and **carry**) and two outputs (**sum** and **carry**). Define an array that implements this function. Write an expression that uses this array to find the sum and carry of **logic** signals **a** and **b**.

Expressions

Combinational logic is defined using expressions as in the example above. Expressions include operators that operate on “operands” – numeric literals (“constants”), **logic** variables and arrays of these.

- Operators with higher “precedence” are applied before those of lower precedence.
- If two operators are of equal precedence they are applied from left to right¹.
- Operands and operators affect the number of bits in the result (the “width”) as described below.
- Values are truncated or padded (with zeros) on the left to the width of the final expression when computing the result

To evaluate an expression:

1. find the value and width of each operand (signals or literals)
2. group operators with operands according to precedence (or left to right when the precedence is the same)
3. apply the operators, obeying rules for padding and truncation as appropriate for the operands

The descriptions below specify the width of the result of each operator. The expression is then evaluated using the width of the largest result, including the left-hand side of an assignment.

¹Except the conditional operator: **a ? b : c ? d : e** is evaluated as **a ? b : (c ? d : e)**

Operators

The following describe some useful Verilog operators in order of decreasing precedence.

Slices A range of bits in a packed array (a “slice”) can be extracted using a range of indices in brackets (*[first:last]*) after the array name. The bit order cannot be reversed. The width is the number of bits in the slice.

Negation Logical negation (!) is zero (0) when applied to a non-zero operand and one (1) otherwise. The width is 1 bit.

Bitwise negation (~) inverts the value of each bit. The width is the width of the operand.

Exercise 7: What are the values of the following expressions: `!4'b010?` `~4'b010?` `0+~(!0)?`

Arithmetic Multiplication (*), division (/), addition (+) and subtraction (-) can be applied to arrays. The first two have higher precedence. The width is the largest of the two operands' widths.

Shift Right-(>>) and left-shift (<<) operators shift the bits in the array operand on the left by the amount on the right. The width is the width of the left operand.

Comparison When arithmetic comparison operators (<, >, <=, >=) are applied to arrays the result is 1 if the comparison is true and 0 otherwise. The width is 1 bit.

Equality The result of the equal (==) or not-equal (!=) operators is 1 if the operands are equal and 0 otherwise. The width is 1 bit.

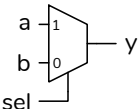
Bitwise Logical Bitwise logical operators are applied to the corresponding bits in two operands. Bitwise-and (&) has higher precedence than bitwise-or (^) which has higher precedence than bitwise-or (|). The width is the largest of the two operands' widths.

Logical Logical operators result in 1'b1 if both operands (for &&) or either operand (for ||) are non-zero. Logical and (&&) has higher precedence than logical or (||). The width is 1 bit.

Exercise 8: An array declared as `logic [15:0] n;` and has the value `16'h1234`. What are the values and widths of the following expressions? `n[15:13] !n~n[3:0] n>>4`
`n + 1'b1 n[7:0] - n[3:0] n >= 16'h1234 n ^ '1 n`
`&& !n n * ( !n + 1'b1 )`

Conditional Operator Verilog's conditional operator is a concise syntax for describing a two-way multiplexer. The operator consists of three parts: the condition, the true value and false value. The result of the operator is the true value if the condition is non-zero, or the false value otherwise. The width is the largest of the true and false value widths.

For example:

`assign y = sel ? a : b` ⇔ 

sets `y` to `a` when `sel` is non-zero and sets `y` to `b` when `sel` is zero.

Exercise 9: What are the width and value of the expression: `3 ? 16'd10 : 8'h20?` If `x` has the value 0, what is the value of the expression: `x ? 1'b1 : 1'b0 ?` If `x` has the value -1?

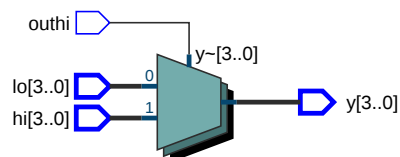
The following example implements a multiplexer that selects from one of two 4-bit inputs:

```
module ex36 (input logic outhi,
             input logic [3:0] hi, lo,
             output logic [3:0] y) ;

    assign y = outhi ? hi : lo ;

endmodule
```

which results in:



Conditional operators can be used to concisely describe trees of multiplexers. Unlike other operators, a sequence of conditional operators is evaluated from right to left.

Exercise 10: Draw the schematics corresponding to:

```
y = a ? ( b ? s1 : s2 ) : ( c ? s3 : s4 );
y = a ? s1 : b ? s2 : c ? s3 : s4;
y = a ? b ? c ? s3 : s4 : s2 : s1 ;
```

Concatenation The concatenation operator ({,}) combines expressions into a wider value. The width is the sum of the widths.

Exercise 11: Use slicing and concatenation to compute the byte-swapped value of an array `n` declared as `logic [15:0] n`.

Exercise 12: If *n* has the value 16'h1234, what are the value and width of: {*n*[7:0], *n*[15:8], 4'b1111}?

Exercise 13: Use concatenation to shift *n* left by two bits.

Concatenations of variables can also be used on the left hand side of an assignment.

Exercise 14: Use concatenation to assign the high-order byte of *n* to *a* and the low-order byte to *b*.

Statements

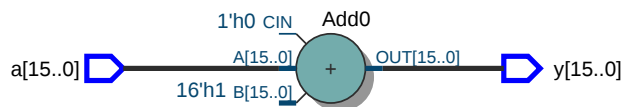
Modules contain variable declarations followed by **assign**, **always_ff**, and component instantiation statements. Statement order does not matter – the hardware created by each statement operates simultaneously (*concurrently*).

assign

The **assign** statement continuously assigns the value of the expression on the right-hand side (RHS) to the signal on the left-hand side (LHS). The most significant bits are dropped if the RHS is wider than the LHS. For example, this module:

```
module ex62 ( input logic [15:0] a,
              output logic [15:0] y ) ;
    assign y = a + 1 ;
endmodule
```

results in this logic:



Exercise 15: Some software warns about truncation. How could you re-write the **assign** statement to avoid such a warning?

always_ff

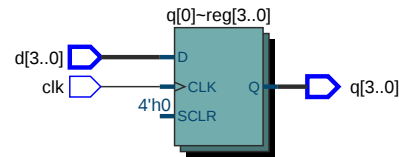
The **always_ff** statement creates registers and flip-flops. For example, the following Verilog:

```
module ex2 (input logic [3:0] d,
            input logic clk,
            output logic [3:0] q) ;

    always_ff @(posedge clk)
        q <= d ;

endmodule
```

synthesizes a 4-bit D flip-flop that transfers the *d* input to the *q* output on the rising (positive) edge of *clk*:



Note the use of “non-blocking assignment” (**<=**) in **always_ff** statements. As with the blocking assignment (**=**), the leftmost bits are dropped if the RHS is wider than the LHS.

Exercise 16: Write an **always_ff** statement that toggles (inverts) its one-bit output on each rising edge of the clock.

Component Instantiation

Module instantiation inserts a copy of (“instantiates”) one module into another and connects signals to the instantiated module’s input and output ports. For example, the module:

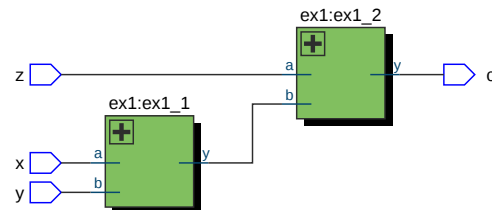
```
module ex60 ( input logic x, y, z,
              output logic c ) ;

    logic t ;

    ex1 ex1_1 ( x, y, t ) ;
    ex1 ex1_2 ( z, t, c ) ;

endmodule
```

creates a 3-input AND gate by instantiating two instances of the *ex1* module defined earlier:



Exercise 17: Identify the following in the diagram above: component names, component “instance names,” component port names, module port names. Label the signal *t* in the schematic.

Exercise 18: Rewrite the *ex60* module using operators. Which version – “structural” or “behavioural” – is easier to understand?