

## More System Verilog

This lecture describes Verilog types, operators and the evaluation of expressions as well as the most common Verilog statements. More details are available in the System Verilog standard, IEEE Std 1800-2012.

After this lecture you should be able to: predict the size and value of a Verilog expression that uses the signals, constants and operators described below; and predict the flow of control between the statements described below.

Revision 2 (added example of module instantiation)

### More Syntax

---

#### Reserved Words

System Verilog has about 250 reserved words that may not be used as module or signal names. Using one of these (e.g. **buf**, **time**, **wait**, **disable**, **reg**, **table**, **input**, ...) will result in a syntax error. An editor with syntax highlighting is a convenient way to identify and avoid reserved words.

#### Types and Values

Verilog's **logic** signals can use four values: **0** (false), **1** (true), **x** (unknown) and **z** (high impedance) to model the operation of logic circuits. A **z** (high-impedance) value is used for synthesis of tri-state outputs. The **x** (unknown) value is used in simulations to indicate an unknown value.

The **bit** and **integer** types are “two-state” (0 and 1) are used as array indices to describe circuits with repeated elements.

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

#### Signedness

Signals can be declared **signed**. Based literals are unsigned unless **s** is used with the base (e.g. **4'shf**). Negative values are in twos-complement format.

Signedness does not affect the values of the bits, only how some operators treat the sign bit.

#### Width and Sign of Expressions

Figure 1<sup>1</sup> shows how the size of an expression depends on its operands (signals and literals and other

expressions). The notation  $L(\cdot)$  refers to the length (width) of an operand.

Values are left-padded or left-truncated as necessary. Padding replicates the sign bit only for signed values.

The result of an expression is unsigned unless all the operands are signed.

#### Arrays

---

Variables may have (multiple) “packed” and “unpacked” dimensions.

**Packed** dimensions appear before the signal name. These bits are stored contiguously (“packed”) and the packed item can be treated as a scalar – a single number – in expressions. Packed dimensions typically model a word or bit fields within a word. For example, **logic [3:0][7:0] ax** ; would describe a 32-bit word composed of 4 bytes of 8 bits and **ax[3]** would be the most-significant byte and **ax[0][7:4]** would be the most-significant nybble of the least-significant byte.

**Unpacked** dimensions appear after the signal name. These bits need not be stored contiguously. Unpacked arrays model memories. In these only one element can be accessed at a time. For example: **logic [7:0] rom [32]** ; would model a 32-byte memory.

In array references, the unpacked dimension(s) 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.

#### Array Literals

Array literals (constants) can be defined by grouping the individual elements within **{...}**. The quote

---

<sup>1</sup>Tables are from the IEEE System Verilog Standard, IEEE Std 1800-2012.

**Table 11-21—Bit lengths resulting from self-determined expressions**

Expression	Bit length	Comments
Unsigned constant number	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % &   ^ ^~ ~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: === != == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: &&    -> <->	1 bit	All operands are self-determined
op i, where op is: & ~&   ~  ^ ~^ ^~ !	1 bit	All operands are self-determined
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i, ..., j}	L(i)+..+L(j)	All operands are self-determined
{i{j, ..., k}}	i × (L(j)+..+L(k))	All operands are self-determined

Figure 1: Size of Expressions.

distinguishes array literal syntax from the syntactically similar concatenation operator (below).

### Examples

The examples below illustrate the rules described above.

```

module ex12 ;
  initial begin

    logic [3:0] x ;
    logic signed [15:0] y ;
    logic [3:0] [7:0] z [15:0] ;

    // what are the values of x?
    x = 4'b01xz ;
    x = -1 + 0 ;
    y = -1 + 4'shf ;
    x = y ;

    // what are the values of z?
    z[0] = '1 ;
    z[0] = {4{4'b1}} ;
    z[0][0][7] = 1 ;
    z[15:0] = '16{z[0]} ;

  end
endmodule

```

**Exercise 1:** What are the packed and unpacked dimensions of each

declaration?

**Exercise 2:** What are the signedness, size and value of each constant and each expression above?

### Operators

Figure 2 below lists the Verilog operators and Figure 3 their precedence. Operators that differ from those in C and that are widely supported for synthesis are described below.

**Arithmetic vs Logical Shift** Left shift always zero-fills on the right. Arithmetic right shifts (>>) replicate the sign bit if the result is signed. Logical right shifts (>>) always zero-fill.

**Logical Reduction Operators** These unary (one operand) operators apply a logical operation to the bits of the operand. For example, to test if any bit is set we can apply the or-reduction operator.

**Conditional Operator** The result when the conditional expression contains x or z is not what you

**Table 11-1—Operators and data types**

Operator token	Name	Operand data types
=	Binary assignment operator	Any
+= -= /= *=	Binary arithmetic assignment operators	Integral, <b>real</b> , <b>shortreal</b>
%=	Binary arithmetic modulus assignment operator	Integral
&=  = ^=	Binary bitwise assignment operators	Integral
>>= <<=	Binary logical shift assignment operators	Integral
>>>= <<<=	Binary arithmetic shift assignment operators	Integral
?:	Conditional operator	Any
+ -	Unary arithmetic operators	Integral, <b>real</b> , <b>shortreal</b>
!	Unary logical negation operator	Integral, <b>real</b> , <b>shortreal</b>
~ & ~&   ~  ^ ~^ ^~	Unary logical reduction operators	Integral
+ - * / **	Binary arithmetic operators	Integral, <b>real</b> , <b>shortreal</b>
%	Binary arithmetic modulus operator	Integral
&   ^ ^~ ~^	Binary bitwise operators	Integral
>> <<	Binary logical shift operators	Integral
>>> <<<	Binary arithmetic shift operators	Integral
&&    -> <->	Binary logical operators	Integral, <b>real</b> , <b>shortreal</b>
< <= > >=	Binary relational operators	Integral, <b>real</b> , <b>shortreal</b>
=== !==	Binary case equality operators	Any except <b>real</b> and <b>shortreal</b>
== !=	Binary logical equality operators	Any
==? !=?	Binary wildcard equality operators	Integral
++ --	Unary increment, decrement operators	Integral, <b>real</b> , <b>shortreal</b>
<b>inside</b>	Binary set membership operator	Singular for the left operand
<b>dist</b>	Binary distribution operator	Integral
{ } { }	Concatenation, replication operators	Integral
{<<{ } } {>>{ } }	Stream operators	Integral

Figure 2: Verilog operators.

might expect (the rules are complex).

**Equality** Equality comparison (==, !=) returns x if either operand contains x or z.

Wildcard equality (==?, !=?) excludes bits that are x or z in the right operand from the compar-

ison. The result is always 0 or 1.

Four-state comparison (===, !==) compares the two operands for an exact match, including x and z.

**Concatenation** Bits can be concatenated by sepa-

**Table 11-2—Operator precedence and associativity**


Operator	Associativity	Precedence
() [] :: .	Left	Highest  Lowest
+ - ! ~ & ~&   ~  ^ ~^ ^~ ++ -- (unary)		
**	Left	
* / %	Left	
+ - (binary)	Left	
<< >> <<< >>>	Left	
< <= > >= inside dist	Left	
== != === !== ==? !=?	Left	
& (binary)	Left	
^ ~^ ^~ (binary)	Left	
(binary)	Left	
&&	Left	
	Left	
?: (conditional operator)	Right	
-> <->	Right	
= += -= *= /= %= &= ^=  = <<= >>= <<<= >>>= := :/ <=	None	
{ } { }	Concatenation	

Figure 3: Operator Precedence.

rating expressions with commas and surrounding them with braces ({}).

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

Arrays can also be spliced back together with the concatenation operator ({}). For example, we can swap the bytes of a 16-bit word **b** using: {b[7:0],b[15:8]}.

**Replication** The syntax is similar to concatenation but uses two pairs of nested braces and repetition value.

**Array Slices** The array subscript operator can be used to extract contiguous portions (slices) of an array. The bit order cannot be reversed.

Part of a packed array (a “slice”) can be referenced with a range of indices as shown above.

**Cast** Although not an operator, a cast (') can be used to change the type, size or signedness of an expression.

**Examples**

```

module ex13 ;
  initial begin

    logic [15:0] x ;
    logic signed [15:0] y ;

    x = 16'hfff0 ; // x=65520
    y = x >>> 1 ; // y=0x7ff8
    y = signed'(x) >>> 1 ; // y=0xffff8
    y = |y ; // y=1

    x = 8'h4x ; // x=X
    y = x == 8'h4x ; // y=X
    y = x[6:3] === 7'b100x ; // y=1
    y = x ==? 8'h4x ; // y=1
    y = {x[7:4],x[6]} ; // y=0x0009
  end
endmodule
    
```

## Modules

Small, simple circuits are easier to design and verify than large ones. For this reason it's good practice to divide designs into smaller parts<sup>2</sup>. In some cases a large part can be constructed from multiple copies of the same part. And if the partitioning is done carefully, it's often possible to re-use the parts in other designs. 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 a hardware design 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.

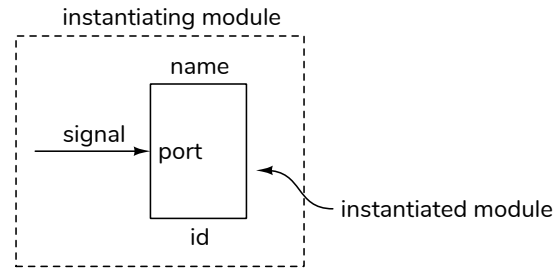
Here's an example of a module named **bit\_store** that defines an **nb**-bit register:

```
module bit_store
  #(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
```

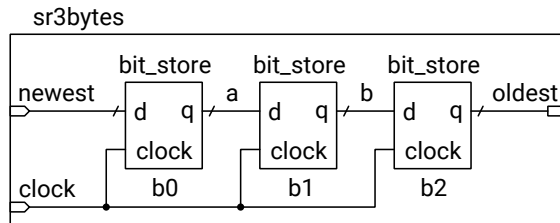
The parameter **nb** has a default value of 1 which is used if a value is not given when the module is instantiated. There are two input ports (named **d** and **clock**) and one output port (named **q**).

To instantiate one module into another it's necessary to provide the name of the module, an instance name (to identify individual instances of the same module), and the correspondence between signals in the instantiating module and the ports in the instantiated module:

<sup>2</sup>When should you stop dividing a design into smaller parts? A good rule of thumb is to make sure each part can be described on a single page.



For example, an 8-bit 3-stage shift register:



could be defined as follows:

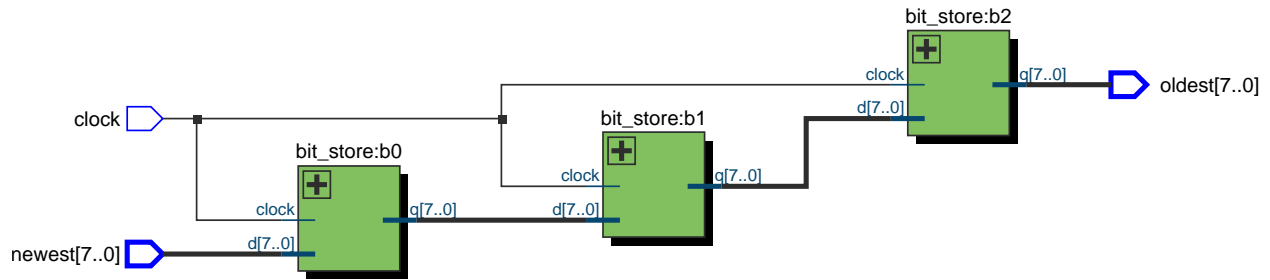
```
module sr3bytes
  (
    input logic [7:0] newest,
    output logic [7:0] oldest,
    input logic clock
  ) ;
  localparam nb = 8 ;
  logic [7:0] a, b ;
  // matching by order
  bit_store #(nb) b0 (newest, a, clock) ;
  // matching by name (order does not matter)
  bit_store #(.nb(nb)) b1 (.q(b), .clock, .d(a)) ;
  // wildcards for names that match
  bit_store #(.nb(nb)) b2 (.d(b), .q(oldest), .*) ;
endmodule
```

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 is as expected:



## Parallel Statements

A module can contain any number of the following parallel statements, all of which execute concurrently.

### always Procedural Blocks

**always** blocks execute the following statement in an infinite loop. Execution of the next statement is often controlled by one of the following:

**#number** delays *number* before each execution. This is not synthesizable but is useful for simulation.

**@(expression)** waits until the value of the expression (the “sensitivity list”) changes. This can be used to model combinational, latched or flip-flop logic.

The type of logic generated by the always block depends on the the sensitivity list and which variables are assigned to within the block.

If for some conditions variables are not assigned to within the block then the language semantics require that memory be generated so that the previous value is retained. This memory can be edge-triggered (when the sensitivity list uses **posedge** or **negedge**) or a latch (otherwise). On the other hand, if all variables in the sensitivity list are updated each time the block executes then combinational logic is generated.

A common mistake is to omit signals from the sensitivity list or not assign to a variable. This results in unintended latched logic.

To avoid this, System Verilog has three variants of the **always** procedural block: **always\_ff**, **always\_comb** and **always\_latch** that document the designer’s intent. A warning or error is generated if

the sensitivity list or assignments within the block would not result in the intended type of logic.

An **always\_comb** does not need a sensitivity list – the implied sensitivity list includes all signals that are ‘read’ within the block.

An **always\_ff** block requires a sensitivity list that includes **posedge** or **negedge** qualifiers on each signal.