System Verilog

Verilog expressions are similar to those in C with additional properties so they can better describe hardware. Verilog has both parallel and sequential statements that allow it to better describe the behaviour of hardware. 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.

Types and Logic Values

Verilog 4-state types, such as `logic`, can have four values: 0 (false), 1 (true), `x` (unknown) and `z` (high impedance). These are used for modeling logic.

2-state types such as `bit` and `integer` can have values 0 and 1. These are primarily used as counters and array indices.

Numeric Constants

In addition to declaring the size and base as described previously, constants ("literals") can also be declared as signed by prefixing the base with 's' (e.g. `4'shf` for a 4-bit -1). Decimal constants are treated as signed by default.

A numeric constant can also include `x` (unknown) or `z` (high-impedance) values. These have useful interpretations for synthesis (don't-care and high-impedance respectively) but when used in simulations the result, in most cases, will be unknown (`x`).

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

Signedness and Size

Variables and expressions can have a size (measured in bits) and can be signed or unsigned. The behaviour of some operators depends on the signedness of the operands.

Signedness does not affect the values of the bits (0 or 1) only how operators act on them. Negative values of signed values are assumed to be in twos-complement form.

Decimal literals are signed and based literals are unsigned unless `s` is used with the base.

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

Figure 1[1] shows how the size of an expression depends on its operands.

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

Arrays

Variables may have multiple "packed" and "unpacked" dimensions.

Packed dimensions are those given 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 (e.g. a 32-bit word composed of 4 bytes of 8 bits).

Unpacked dimensions appear after the signal name. These bits may not necessarily be stored contiguously. Unpacked dimensions model memories where only one element can be accessed at a time.

In array references, the unpacked dimension(s) are specified first, followed by the packed dimensions (if any).

Array Literals

Array literals (constants) can be defined by grouping the individual elements within `'{...}`. The quote distinguishes array literal syntax from the syntactically similar concatenation operator.

[1]Tables are from the IEEE System Verilog Standard, IEEE Std 1800-2012.

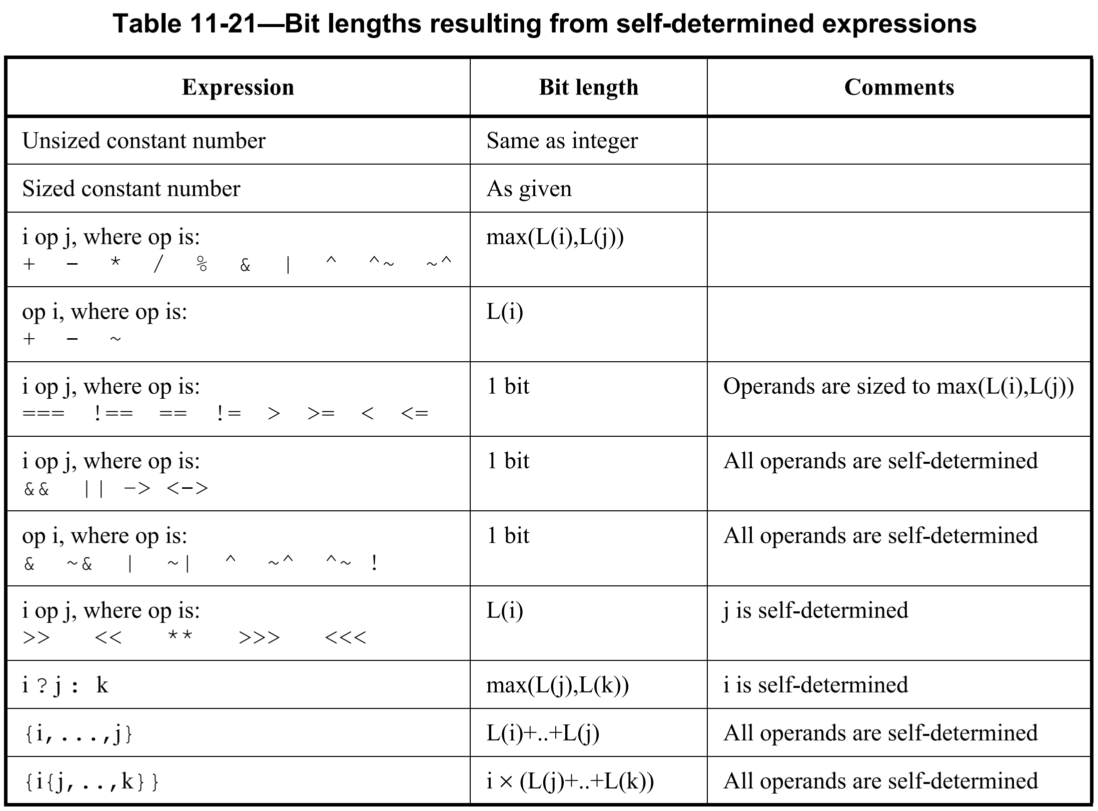
Table 11-21—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsized 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.

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] ;

      x = 4'b01xz ; //
      x = -1 + 0 ;  // x=15

      y = -1 + 4'shf ;  // y=-2
      x = y ;           // x=14

      z[0] = '1 ;         // z[0]=4294967295
      z[0] = {4{4'b1}} ; // z[0]=4369
      z[0][0][7] = 1 ;   // z[0]=4497
      z[15:0] = '{16{z[0]}} ;  // z[*]=4497

   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.

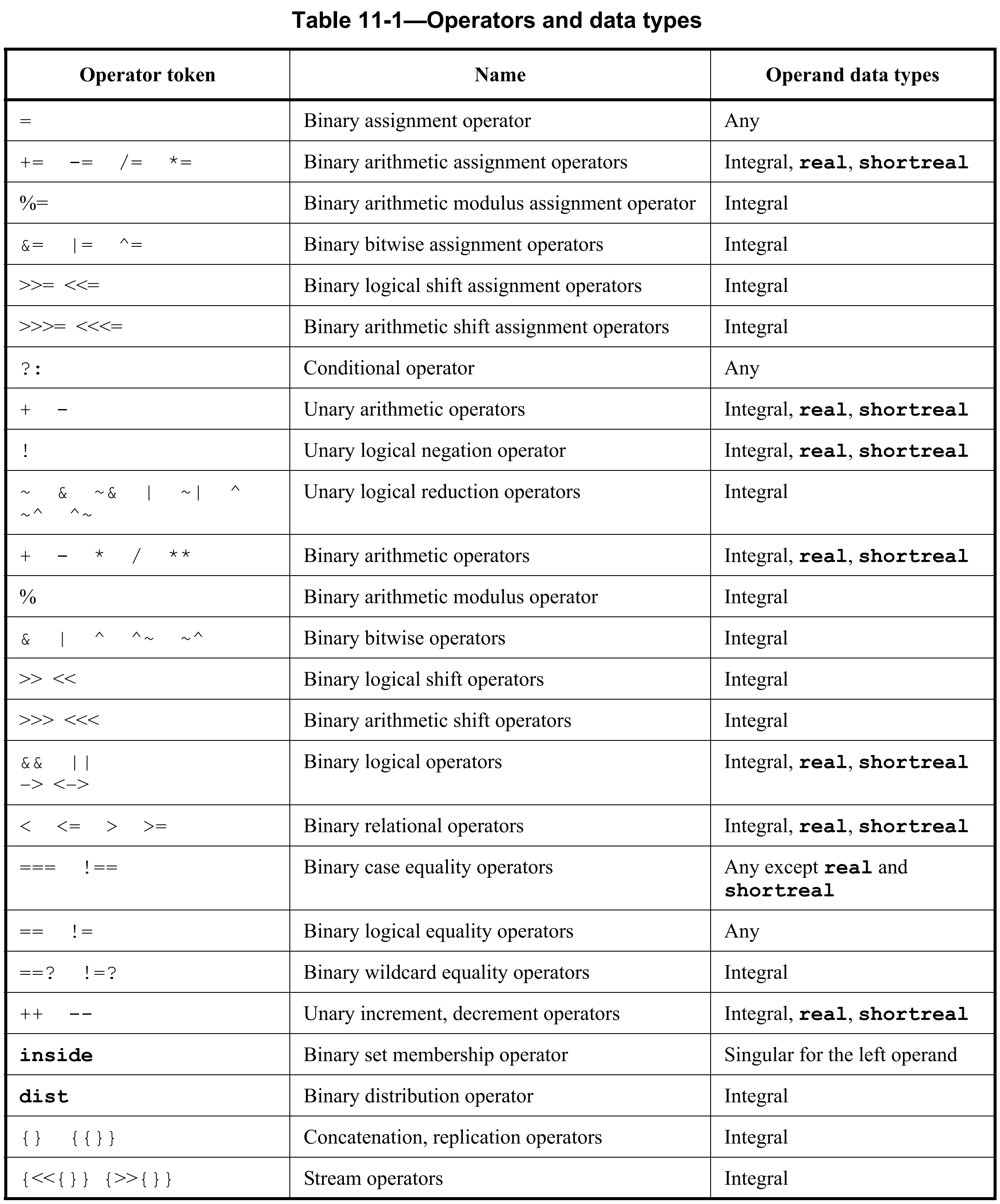

Table 11-1—Operators and data types

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

Figure 2: Verilog operators.

Conditional Operator The result when the conditional expression contains x or z is not what you might expect (the rules are complex).

Equality The regular equality comparison operators (e.g. ==) return x if either operand contains x or z.

The wildcard equality comparison (==?) excludes bits that are x or z in the right operand from the comparison. The result is always 0 or 1.

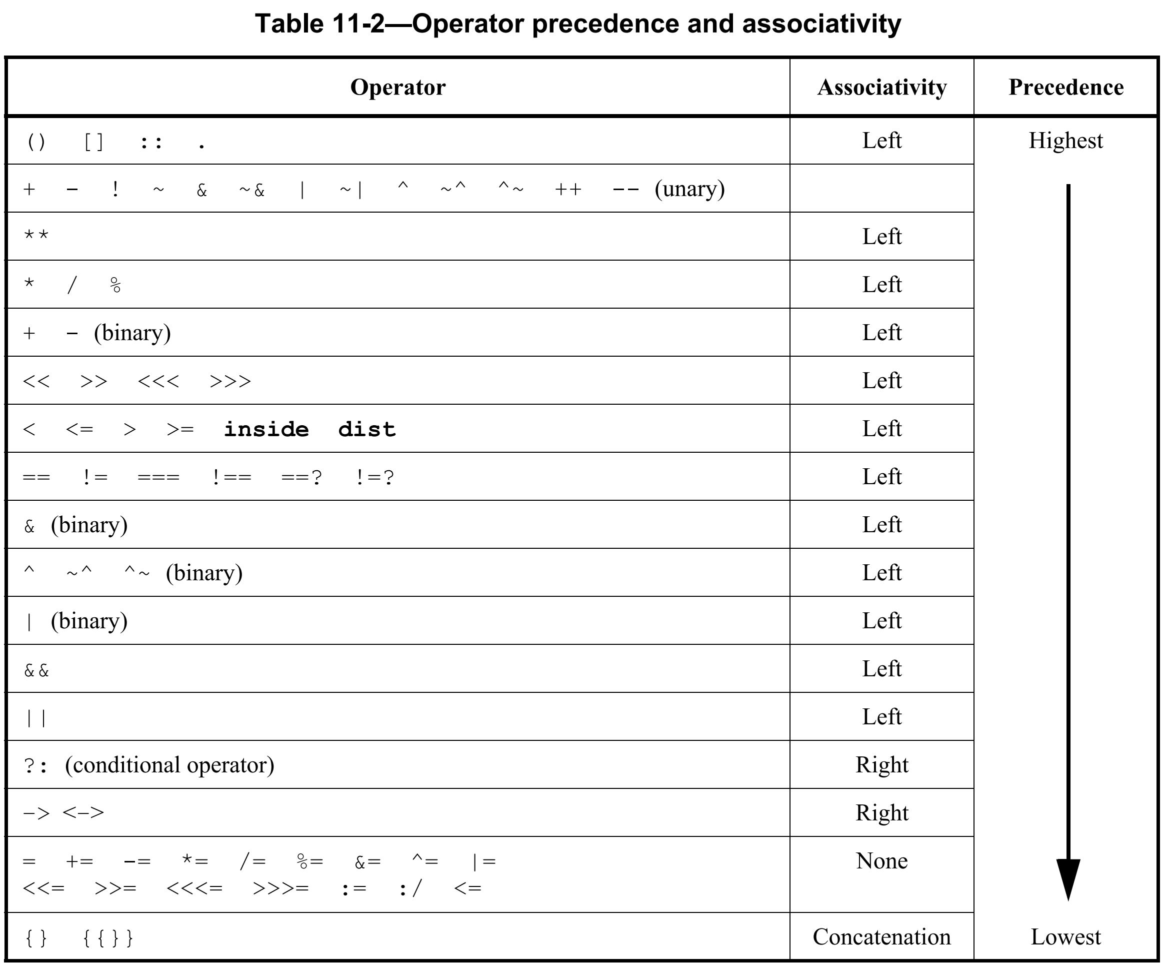

Table 11-2—Operator precedence and associativity

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

Figure 3: Operator Precedence.

The 4-state comparison (===) compares the two operands for an exact match, including x and z.

Concatenation Bits can be concatenated by separating expressions with commas and surrounding them with braces ({}).

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

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.

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=0xfff8
      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

Modules represent the top-level blocks of a hardware description. A module includes declarations and parallel (concurrently executing) statements between `module` and `endmodule`. The module's header defines the module's name, parameters and ports. Ports can be `in`, `out` or `inout` (to model bidirectional signals).

When a module is instantiated, a signal can be connected to module port by port order (`sig`); by port name together with an explicit signal name (`.port(sig)`), by port name with the signal name implicit (`.port`) and using a wildcard (`.*`) that matches all remaining matching port and signal names. Expressions can be used instead of a signal name.

Parameters

Parameters are named constants. They are often included in the module's declaration to allow customization of each instantiation of a module. Parameters such as bus width or clock frequency are common.

Default values can be specified for parameters. As with signals, the values of parameters can be specified by position or by name.

The following example shows how parameters with default values can be specified, how they can be used to specify the dimensions of an array port and how system functions such as `$size()`, `$right()` and `$clog2(x)` ($\lceil \log_2(x) \rceil$) can be used to compute parameter values in module instantiations:

```
module setbits  #(M=7,L=0) (output [M:L] x);
   assign x = '1 ;
endmodule

module ex18 ;
   logic [31:16] x ;
   setbits #(.L($right(x,1)),.M(31)) s0(x);
endmodule
```

`struct`, `typedef` and `enum`

System Verilog includes several data modeling features derived from C that can be used in declarations.

struct structures allow grouping variables (or signals) of different data types. Each is accessed using a dot followed by the member name. In System Verilog structs can be declared `packed`, allowing them to also be treated as scalars, similar to packed arrays.

typedef typedefs define user-defined types.

enum defines enumerated types that can take on one of a set of values.

Functions

Functions can be declared in modules and can be used to model *combinational* logic (only). Functions allow the same logic to be re-used in multiple places. Specialized combinational logic functions (e.g. 7-segment LED decoder) are often coded as functions.

Packages

Packages serve a purpose similar to header (`.h`) files in C. Since function and data type declarations are local to the module in which they are declared, to reuse them across modules they should be declared between `package`/`endpackage` keywords. These functions and data types can then be made available by using an `import` statement in a module:

```
package ex15pkg ;

   typedef enum { INIT=1, ITER=2,
                  DONE=4 } state_t ;

   typedef logic [3:0] nybble ;

   typedef struct {
      logic [7:0] r, theta ; } polar_t ;

   // (slow) 32-bit priority encoder
   function automatic logic [5:0]
     pri(logic [31:0] x) ;
       for ( pri=32 ; pri>0 ; pri-- )
         if ( x[pri-1] ) return pri ;
   endfunction

endpackage

module ex15 ( input logic [15:0] irqs,
              output logic [4:0] irq,
              output logic active ) ;
```

```
    import ex15pkg::* ;

    nybble [7:0] word32 ;

    polar_t pos ;

    assign pos = '{ 8'd255, 8'h7f } ;

    assign irq = pri(irqs) ;
    assign active = irq ? '1 : '0 ;

endmodule
```

The `import` statement (and declarations) can be placed outside and before the module declaration. In this case they apply to subsequent modules that are compiled *with the same command*. This is not recommended because it's not possible to tell from the source which files should be compiled at the same time or in which order.

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.

`initial` Procedural Blocks

An `initial` block is executed once at the start of the simulation. These are only synthesizable when used to initialize FPGA memory and registers.

Continuous `assignment`

An `assign` statement continuously assigns (connects) the result of an expression to a net. It is a more concise way to define combinational logic than using `always_comb` but is limited to a single expression.

Sequential Statements

The following statements appear within `always` or `initial` procedural blocks and execute sequentially (one after the other).

begin/end

These keywords group statements that should be executed together. They are similar to braces in C. They also begin a new scope for declarations. They can be labelled so that any variables declared within the block can be referenced (e.g. by simulators).

for/while/do/repeat/forever loops

The `for`, `while` and `do` loops are the same as in C. The `repeat` and `forever` statements execute a statement a given number of times or forever. The `break` and `continue` statements from C can also be used.

As shown in the first lecture, loops generate combinational logic and are only synthesizable when the number of iterations is known at compile time. However, they are very useful when writing testbenches for simulation.

Blocking and Non-Blocking Assignments

A blocking assignment (`=`) evaluates the RHS (right hand side) and immediately sets the value of the variable on the LHS (left HS).

A non-blocking assignment (`<=`) evaluates the RHS but does not set the value of the LHS until the next time step (typically, after all sequential statements have executed).

For example,

```
a = b ;
b = a ;
```

would set both `a` and `b` to the value of `b` while

```
a <= b ;
b <= a ;
```

would swap the values of `a` and `b`.

Recommended practice is to use non-blocking assignments for sequential logic (in `always_ff` blocks). This models the behaviour of flip-flops whose outputs don't change until the next rising clock edge.

Blocking assignments are more convenient for designing combinational logic (inside `always_comb` blocks) as this matches programming language semantics and allows use of intermediate results.

Assignments can synthesize combinational or sequential logic depending on the sensitivity list and type of the enclosing `always` block as described above.

Do not assign to the same signal from more than one `always` block – the order in which always blocks execute is undefined.

You can sometimes break these rules but your code is more likely to have logical errors and the results may not be repeatable (between simulators or from simulation to synthesis).

if/else

The if/else statement syntax is similar to C and synthesizes multiplexers trees whose hierarchy defined by the order of the conditions.

case/casez

This is the equivalent of C's switch statement. Between `case` and `endcase` are a sequence of expressions, each followed by a colon and a statement.

A `default` value indicates the statement that should be executed if none of the vales match.

By default a case statement synthesizes a multiplexer tree with each condition tested in sequence. Thus the `case` semantics are the same as a sequence of nested `if-else` statements.

Placing `unique` before `case` implies that *exactly one* case expression will match. This allows the expressions to be tested in parallel resulting in faster logic.

Placing `priority` before `case` implies that *at least one* case expression will match and that the first match should be used.

Since both `unique` and `priority` imply that one of the expressions will match, when these are used there should be no `default` expression.

The `casez` variant of the `case` statement allows the case expressions to include `?` (or `z`) values which are treated as "don't care" bits. There is also a `casex` variant where `x` (undefined) also matches anything which is usually not desired.

Interfaces

An `interface` groups signals. Interfaces are defined by declaring signals between `interface` and `endinterface`. A module can then declare an instance of an interface and use it to connect other modules. Individual signals from an interface are selected using the same 'dot' syntax as structure member names. For example, an SPI bus could be modelled as:

```
interface spibus ( input logic clk ) ;
   logic sclk, mosi, miso, ssn ;
endinterface

module master ( spibus io ) ;
   logic x ;
   assign io.mosi = 1 ;
   assign x = io.miso ;
endmodule

module slave ( spibus io ) ;
   logic x ;
   assign x = io.mosi ;
   assign io.miso = 2 ;
endmodule

module ex17 ( input logic clk ) ;
   spibus io ( clk ) ;
   master m0 ( io ) ;
   slave s0 ( io ) ;
endmodule
```

Interfaces can also include type and function declarations, and ports that allow connections to all connections to that interface (e.g. for clock and reset signals).

Interfaces can also define variants (e.g. master and slave) defined as `modports`, that include different combinations of signals, each of which is declared as an input or an output.

Other Language Topics

Variables and Nets, `reg` and `wire`

A *variable* can store a value. It models a register. A variable can only be assigned within an always block. A *net* does not store a value. It models a connection. It must be driven by a continuous assignment statement.

Earlier versions of Verilog used `reg` and `wire` declarations instead of `logic`. Assignments in procedural statements (within `always` blocks) must be to "variables" declared `reg`. Other signals are "nets" and are declared `wire`.

However, `reg` variables need not represent registers and `wire` signals often originate in register outputs. Thus `wire` and `reg` convey little information. Use System Verilog's `logic` declarations instead whenever possible.

Exercise 3: Should each of the following nets (or variables) be declared `wire` or `reg`?

```
module test (a,b,c,d,q) ;
dff d0 (clk,d,q) ; // assume only q is an output
assign d = a & b ;
always@* clk = a & c ;
endmodule
```

Only nets may have multiple drivers (e.g. to model buses with tri-state drivers). There are various flavours (`wire`, `tri`, `wand`, ..) each with different "resolution functions" that combine multiple driving values in different ways. Bidirectional (`inout`) module ports must also be declared with net types.

Initialization

The ability to initialize registers modeled as ordinary ("static") variables depends on the target technology. ASIC registers power up in undefined states and must be explicitly reset. However, most FPGA technologies allow the power-on values of registers to be be included in the FPGA configuration data that is loaded when the FPGA is powered on. In this case initialization of static variables is synthesizable.

Variables and functions can also be declared `automatic` to define semantics similar to those for dynamically allocated variables in conventional programming languages. These variables can be initialized each time the block is executed. For example, `cnt` below will be reset on each rising clock edge:

```
always_ff@(posedge clk) begin
   automatic logic [7:0] cnt = '0 ;
   // ...
end
```

$readmemh/$readmemb

These system tasks allow large blocks of constant data, in hex and binary bases respectively, to be read from files. For example: `$readmemh("data.hex",memory);` would read hex values into the variable (array) `memory` from the file "data.hex."

System Tasks

Functions beginning with $ are system tasks. Some examples: `$display()`, similar to C's `printf()`, can be used to print messages during simulation; `$finish` and `$stop` can terminate a simulation.

Identifiers and Reserved Words

System Verilog has about 250 reserved words that cannot be used as identifiers. Unfortunately, this includes many words that you might be tempted to use as signal names (`buf`, `time`, `wait`, `disable`,...). The language specification has a complete list.