Verilog Statements

Verilog has both parallel and sequential statements that allow it to better describe the behaviour of hardware. This lecture provides more details on Verilog statements. More details are available in the Verilog standard, IEEE Std 1800-2012. After this lecture you should be able to predict the flow of control between the statements described below.

Modules

Modules represent the top-level blocks of a hardware description. A module includes declarations and parallel 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).

Parameters

Parameters are constants. They are often included in the module's declaration to allow customization of each instantiation of a module. Parameters such as bus widths or clock frequencies 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 a vector port and how system functions (e.g. `$right()`) can be use to query array dimensions 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 nets) 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 are used to model combinational logic. Functions allow the same logic to be re-used in multiple places. Functions such as encoders, multiplexers, decoders and arithmetic functions 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 re-use 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 `assign`ment

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.

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

System Verilog `interface` objects can simplify the use of buses. Interfaces are defined by declaring signals between `interface/endinterface`. A module can then declare an instance of this interface and use it to connect other modules. Individual signals of an interface are selected using the same 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 instances of an 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

`reg` and `wire`

Versions of Verilog before System Verilog used `reg` and `wire` declarations instead of `logic`. Assignments in procedural statements (i.e. within `always` blocks) must be to "variables" declared `reg`. Continuous assignments (i.e. `assign`) or outputs from component instantiations must be to "nets" 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 when possible.

Exercise 1: 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."

Reserved Words

System Verilog has about 250 reserved words that cannot be used as identifiers.