This is just a summary of the features of Verilog you might have to use.
In no way is it a complete and exhaustive list of the commands you may
use, but I hope it is enough to get you started. Some of the text is
the same as that in the sample verilog decode program, but I have updated
them slightly. Please let us know if you find errors in this.
possible.

David (s)

Updated 4/29/95 by demon@leland

```
---------------------------------------------------------------------
I. Conditional statements
      1. if (<condition>)
            <statement_1>;
         else
            <statement_2>;
      2. case (<vector>)
            <case_1> : <statement_1>;
            <case_2> : <statement_2>;
            <case_3> : <statement_3>;
            default : <default_statement>;
         endcase
      3. <vector> = (<condition>) ? <value_if_cond_is_true> :
<value_if_cond_is_false>;

II. Bit-wise Logic Operators
      Note: The terms a, b, and c do not have to be single bit terms.
            They may be actually be vectors of several bits.
      1. Logical AND, &
         Examples: a = b & c;
                   a = (b & c) & d & e;
      2. Logical OR, |
         Examples: a = b | c;
                   a = b | (c | d);
      3. Negation, ~
         Examples: a = ~(b & c | d);
                   b = ~c;
      4. Logical XOR, ^
         Examples: a = b ^ c;
                   a = b ^ (c ^ d | e) ^ f;
III. Arithmetic Operators
      Note: You can create vectors out of other vectors by enclosing
            the vectors in braces and seperating them by commas.
      1. Addition, +
         Example: sum = a + b;
                  {carryout, sum[3:0]} = a[3:0] + b[3:0];
      2. Subtraction, -
         Example: diff = a - b;
                  {underflow, diff[3:0]} = a[3:0] - b[3:0];
IV. Assignments
      1. Procedural assignments, always blocks
         Registers are always assigned values procedurally because
         registers store values as opposed to wires which just pass
         value.  Since assignements within an always block only
         connects the left-hand term to the right-hand expression
         for the instant that always block executes, the left-hand
         term must store the value.
         Example:
            module latch(in, clock, out);
            input [3:0] in;
            input clock;
            output [3:0] out;

            wire [3:0] in;
            wire clock; // this statement is not really
                        // necessary because colock is
                        // a single bit
            reg out;

            always @(in or clock)  // Execute whenever in or clock changes
                               // the "or" is not the same as a logical
                               // OR which has the symbol |.
               begin
                  if (clock) // When clock is high
                     out = in; // Left-hand term in an assignment
                             // within an always block must be
                             // a register. The right-hand terms
                             // may be either wires or registers.
               end

            endmodule
```

2. Declarative assignments, assign
   Because the declarative assignments are made continously,
   the left-hand terms of these type of assignments should be
   wires as they don't have to store them. Wires are continuously
   connected to their assigned functions.
   Example:

```
      module complex_black_box(input1, input2, out1, out2);
      input [2:0] input1;
      input [2:0] input2;
      output [2:0] out1;
      output [2:0] out2;

      wire [2:0] input1;
      wire [2:0] input2;
      wire [2:0] out1;
      wire [2:0] out2;

      /* Left-hand term of the assignment must be a wire.
         Right-hand terms may be either wires or registers.
      */
      assign out1 = input1 + ~input2;
      assign out2[2] = input1[2] & input2[0];
      assign out2[1] = ~(~input1[1] + input1[0]);
      assign out2[0] = (input1[2]&input2[2])|(input2[1]);

      endmodule
```

V. Miscellaneous

1. Comments, // or /* */
   You can create comments just like in C or C++ as you probably
   guessed from the examples above. If you use the // method then
   what ever else follows that symbol will be ignored.
   Example:

```
      a = b + c;  // This is an example comment
```

   If you use the /* */ method then the text beginning from the /*
   will be ignored until the first */ is encountered.
   Example:

```
      a = b + c;      /*  This is an example of
                      second method of creating
                      a comment.
                  */
```

2. Defining constants, `define <const_name> <x'yz>
   x stands for the number of binary bits in the contant
   y stands for type of notation for z.
      The different types of notation are:
      h - hexadecimal notation, 0-f
      d - decimal notation, 0-9
      o - octal notation, 0-7

```
            b - binary notation, 0-1
      z stands for the actual value to be substituted
            - An underscore may be place within the "digits" in z to
            enhance readibility. However, the underscore may not be
            the first character of z.

      e.g. To define LENGTH as 8-bit number with the value of 127:
            hexadecimal:    `define LENGTH 8'h7f
            decimal:        `define LENGTH 8'd127
            octal:          `define LENGTH 8'o177
            binary:         `define LENGTH 8'b0111_1111

       Note: You should not place a semi-colon after the define statements.
           If you do place a semi-colon then the following substitution
           will occur:

                  the statement: `define NUM 3'b101;
         will change this: out = `NUM + 3'b111;
                      to this: out = 3'b101; + 3'b111;

           which will give you a syntax error!
   3. Creating delay, #<num_units_of_time>
      Example:
         initial
            begin
               inputA = 3'b001;
               #100        // delay 100 units of time
               inputB =  3'b101;
               #323        // delay 323 units of time
            end
VI. Builtin functions
   1. Displaying a message one time, $display
      This will display statements in text window.  The $display
      function is similar to the printf function in C except it
      automatically generates a newline at the end of a message.
      The letter after the percent sign "%" tells the $display
      how to represent the number to be inserted.
            d - decimal notation
            h - hexadecimal notation
            o - octal notation
            b - binary notation
      The variable $time holds the current time value.
      Example:
         if (flag)
               $display("flag is now %b at time = %d",flag,$time);
   1. Displaying messages continuously, $monitor
      This function is similar to $display except it displays
      messages continuously.  In other words it will display
      statements in text window whenever there is a change with
      one of the variables in the parameter list. It should be
      called at the beginning of the simulation.  The letter
      after the percent sign "%" tells the $monitor how to
      represent the number to be inserted.
            d - decimal notation
            h - hexadecimal notation
            o - octal notation
            b - binary notation
      The variable $time holds the current time value.
      Example:
         initial
            begin
               $monitor("time = %d num = %h",$time, num);
            end
   3. Displaying waveforms, $gr_waves, $gr_addwaves
```

These statements add waveforms to WAVES window.
Example:

```
$gr_waves("mycode",mycode,"mctl0",mycontrol[0],"mctl1",mycontrol[1]);
$gr_addwaves("mctl4",mycontrol[4],"mctl5",mycontrol[5]);
$gr_addwaves("mctl6",mycontrol[6],"mctl7",mycontrol[7]);
```

Or an alternate way with mycontrol shown as a single hexadecimal waveform would be:

```
$gr_waves("mycode",mycode,"mycntl",mycontrol);
```

To look at wires/registers inside a module, the module name is attached to the beginning of the wire with a period seperating them.

eg. To see the wire/register total that exists only internally within the module alu:

```
$gr_waves("a.ttl",alu.total);
```

To print the waves window to a postscript file, use the command:

```
$ps_waves("<filename>.ps");
```

This can be entered either from the verilog interactive mode, or as a line in a module.

3. Stop simulation, $stop
   This is used to create breakpoints within a simulation.
   If you don't have one in your simulation then verilog will exit right after the end of simulation without pausing.

4. Continue simulation, .
   Just type a period and hit return at the prompt to continue with a simulation.