VERILOG "MINI-REFERENCE"

A. BASIC STRUCTURE OF A VERILOG DESIGN

1. Each verilog file is organized into one or more "modules", which may contain I/O line definitions, hardware descriptions and simulation control information.

```
                    module name;
                    ...            /* list of statements */
                    endmodule
```

2. Within a module, both concurrent and sequential behavior can be represented. All concurrent statements/blocks are executed at the same time. Sequential statements are executed singly in the order specified.

   CONCURRENT STATEMENTS:

```
a. Concurrent net assignment:      assign a = c & d;
b. Module instantiation:           multiply  m(out, in1, in2);
c. Gate-primitive instantiation:   and  #10  a1(out, in1, in2);
d. User-defined primitive inst.:   decoder decd1(out, i0, i1);
e. Initial and Always blocks:
     initial                  always
       begin                    begin
          ...                       ...  /* list of sequential stmts */
       end                  end
```

   Note: Initial block executed once at time 0 of simulation.
          Always block executed continuously.

```
     always
          #10 clock = ~clock;  /* clock of period 20ns */
```

   SEQUENTIAL STATEMENTS: execute in order specified by algorithm
                          and delimited by begin/end.

```
     begin          /* enter block at time T */
          #10 a = 0;/* a changes to 0 at time T+10 */
          #10 a = 1;/* a changes to 1 at time T+20 */
     end
     begin
          if (select == 1)
               a = b0;
          else
               a = b1;
     end
```

C. BASIC OBJECTS IN VERILOG (Declared at beginning of module)

1. Registers - store data between assignments (abstract data stores)

```
reg [15:0] pc, acc;  /* two 16-bit registers - bit 15 most sig. */
reg clk;              /* one-bit register */
...
pc = 'h23fa;   /* assign value to pc[15:0] */
clk = ~clk;    /* complement clk */
```

2. Memory - special case of register => a set of registers:

```
reg [7:0] mem1 [0:255];   /* mem1 is a 256-byte memory */
...
mem1[0] = 'b01010101;     /* write to address 0 of mem1 */

Note:  reg [7:0] r;       /* one 8-bit register */
       reg r [7:0];       /* eight 1-bit registers */
```

3. Nets - represent wires (one driver) and buses (multiple drivers).

```
wire w1, w2;         /* single wires */
wire [15:0] w3;      /* group of 16 wires */
tri [15:0] dbus;     /* tri-state bus; no conflict resulution */
trior [0:7] bus2;    /* tri-state bus; OR logic resolves conflicts */
wor [0:7] bus1;/* wired-or bus */
  (also wand and triand)
supply0 gnd;         /* driven by 0 supply */
  (also supply1)
tri vectored [31:0] d1; /* can only update entire bus */
tri scalared [31:0] d2; /* can update individual lines */
```

4. There are 4 "logic values" (0, 1, x, z) and 8 possible signal strengths (we will use the default strength values) that can be assigned to each net/register.

```
0,1 => standard logic values
x   => undefined value
z   => floating (tristate) condition (no active driver)
```

D. PREDEFINED PRIMITIVE LOGIC GATES

1. Types: and, nand, or, nor, xor, xnor, not, buf (standard gates)
   bufif0, bufif1, notif0, notif1 (3-state drivers)

2. Connections for the standard gates are specified in a list, beginning with the ouput: and (out, in1, in2, ...);

3. Connections for the 3-state drivers: bufif0 (out, in, control);
   bufif0/notif0 - active low control input (other two active high)

4. Instantiation: list type, delay and one or more gates & connections.

```
      and #10
              a1(out1, ina1, ina2, ...),     /* gate name is a1 */
              a2(out2, inb1, inb2, ...),     /* gate name is a2 */
              ...
              an(outn, inn1, inn2, ...);     /* semicolon after last */
```

Note: delay parameter can be more complex if desired (see manual).

E. BEHAVIORAL MODELING (I.E. NO IMPLIED STRUCTURE)

1.Procedural assignments: update register value when executed.

```
          r = 0;                      mem[addr] = 'h5b;
          r[1] = 1;                   {cy,acc} = reg1 + reg2;
          r[3:0] = 'b0101;                \_concatenated registers
```

2.Conditional assignment: standard if-else language construct.
(else and else if clauses optional)

```
          if (expression)
            ...     // statement or block
          else if (expr2)
            ...     // statement or block
          else
            ...     // default statement or block
```

3.Multi-way decision (C, Pascal, Ada "case" statement).

```
          case (rega)
               'd0:  result = a+b;      // add if rega = 0
               'd1:  result = a-b;      // sub if rega = 1
               'd2:  result = a&&b;     // and if rega = 2
               ...
               default result = 'bxxxxxxxx; /* default is unknown */
          endcase
```

4.Loop-control statements allow repeated execution of statements.
(primarily C-language structures)

```
     Do "r1" times:      Do while r2 not 0:  Do under control of r:
     repeat (r1)         while (r2)     for (r=1; r<10; r=r+1)
          begin          begin          begin
          ...                 ...                 ...
          end                 end                 end
```

F. SPECIFYING EVENT TIMES FOR PROCEDURAL STATEMENTS

1. Event can take place after a specified delay:

```
        #10 a = b;// in 10 time units read b and update a
        a = #10 b;     // read b now and update a in 10 time units
```

2. For sequential statements, the delay is relative to the execution time of the previous statement (i.e. delays add). The delays of concurrent statements are all relative to the beginning of the set of statements.

```
Starting at time T:
begin     update at:fork         update at:
   #10 a = 0;   (T+10)          #10 a = 0;  (T+10)
   #10 a = 1;   (T+20)          #10 b = 1;  (T+10)
   #10 a = 0;   (T+30)          #10 c = 0;  (T+10)
end                           end
```

3. Timing control related to other events:

```
        @r  a = b;      // update when r changes
        @(posedge r) a = b; // update on positive edge of r
        @(negedge r) a = b; // update on negative edge of r
        @(posedge r or posedge s) a = b;  // update on either pos. edge
          [can also use 'and']
```

G. CONTINUOUS ASSIGNMENTS

1. Continuous assignments are made to nets, rather than to registers, to simulate the driving of nets by one or more gates or drivers. An assignment takes place whenever any of the drivers changes.

```
wire a;
...
assign a = b + c;  /* update net a for any change in b or c */
```

2. Continuous assignments can also be specified in a net declaration.
   Ex: Update bus e 10 time units after a change in either f or g:

```
     wire [7:0] #10  e = f ^ g;
```

H. MISCELLANEOUS CONVENTIONS

1. Comments are enclosed in /* ... */
   Comments at the end of a line follow //...

2. Number formats:

	Unsized	Sized	
decimal:	659	5'd3	(5-bit)
hexadecimal:	'h837ff	16'hz	(16-bit high-z)
binary:	'b010101	4'b0101	(4-bit)
	'b0101_1100_0110	(use _ for clarity)	

3. Identifiers, strings, real, integer values follow C-language syntax.
   Likewise, expressions follow C syntax (mostly).

4. Operators for use in expressions:

```
Arithmetic  add:  +     mult:  *
            sub:  -     quot:  /      remainder:  %
Logical connectives -  and:  &&     or:  ||      not:  ~
Bit-wise logical ops - complement: ~   and: &   or: |  xor: ^
Shift ops:  <<  >>   (ex.  a = b << 2)
Relational ops:  <, >, <=, >=,   ==,!=,   ===, !==
                                   \___\_     \____\_ x and z values
                                         |             must match.
                                 Result is x if ambiguous.
Conditional:   expression ? true_action : false_action
     (do true_action if expression non-zero, otherwise false_action)
Concatenation:  {...}   Ex:  {cy, acc} = a + b;
```