```
============================
VERILOG GUIDE
by NAVID MALEK
nmalek@ce.sharif.edu
============================


variables in verilog:
-reg (can have vector identity)
-integer
-real
-time,realtime

assignment: beceause they are not wires, they should be assigned only in
sequential blocks(always,initial,task,funcion)
read: can be read anywhere (i.e. a driver for a wire)

-------------------------------------------------------------------------
-
variables and wires default is X
-------------------------------------------------------------------------
-
integer : 32bit int-default is X
real : 64bit floating point-default is 0.0
ONLY used in test bench
-------------------------------------------------------------------------
-
time variable:
time my_time;
my_time =$time;// get current time of simulation
-------------------------------------------------
vector:
e.g.
           reg [3:0]A = 3'b 1001;
array:
e.g.
           reg A[3:0] = 3'b 1001;
-----------------------------------------------------------
multi-dementional wire and reg:
e.g.
      reg [7:0] var [1:10][1:100];
      reg   [7:0] mem [1023:0];
-------------------------------------------------
a value is considered as X if all bits are 0 and X or 0 and Z or 0 and x
and z

-------------------------------------------------------------
concatination:
size should be known!
           catx = {a,b,c}
           caty = {a,2'b01,c}
      WRONG:
                 catz = {b,1}

      replication:
           catr = {{4{a}},b,{2{c}}}    // {numOfReplicate{var to
replicate}}
      ------------------------------------------------
bitwise:
           & , | , ^ , ~ , ~^ or ^~ , ~| , ~&
```

```
            //Perform  a bit-by-bit comparison.If input is N-bit, then
output is N-bit.

logical:
           && , || , !

                //Outcome is a single bit: 1, 0, X

                  When inputs are bitvectors:
                      Effect will be to reduce-OR operands, then
                    perform a bitwise or/and/not corresponding to
                    logical operation
                          Example
                            •(4'b1100) && (4'b0011) = 1
                            •!(4'b100x) = 1
rational:
            a < b, a > b, a <= b, a >= b

            Result is 1'b1 if true, 1'b0 if false, 1'bx if either a or b
x
            signed comparison requires both a and b to be
signed,otherwise unsigned comparison
            example:
            4'bs1100 < 4'bs0111) but (4'b1100 > 4'b0111)


case:
both used in the test bench (see e.g.)
           === //case equality
           !== // case inequality

          a === b, a !== b tests a case equality
              will always be 1 or 0
              will include X's and Z's in the comparison
                4'b101X === 4'b101X


     e.g.
           'b1x1 >= 0 ===> x   //becuase X is not only 1 or 0 , it shows
something is wrong ( a gate is out of use)

          a==b, a != b tests a logical equality
              will be 1 or 0 when a and b are fully known
              when any bit of a or b is X, then the result is X
                    4'b101X != 4'b101X
shift operands:
              Logical shift-left <<, Logical shift-right >>
                  Arithmetic shift-left <<<, Arithmetic shift-right >>>

Ternary operator:
                             ? :
              Shorthand notation for a multiplexer
              assign q = c ? a : b;
              c is or-reduced.
                  If result is 1, then q = a.
                  If result is 0, then q = b.
                  If result is x, then q is combined, bit by bit, from a
and b.
```

```
------------------------------------------------------------
continues assignments are done in parrallel
using: assign keyword

a driver for a wire
used a lot in gatelevel modeling

!!!!!!! do not cause a logical loops !!!!!!!!
------------------------------------------------------------
module instantiation:

module_name
//if have parameters
#(
      .parameter_name(value),
      .parameter_name(value)
)
      instance_name
(
      .port_name(connection),
      .....
      .port_name(), //unconnected port
      .port_name(connection)
);
------------------------------------------------------------
input and output rules of module:(PORT BINDING)
input:     reg or net ----->>> net
output: reg or net ----->>> net
inout:     net ----->> net

------------------------------------------------------------
Initial Blocks:
Starts at the beginning of simulation
Exit after last statement
Are done in parallel but not completely parallel. //unknown order
!!!!!!!!!!!!!!!!!! CAN ASSIGN ONLY VARIABLES not
nets(wires)!!!!!!!!!!!!!!!!!!
Use only blocking assignment
since they are not hardware,they are used for test benches

------------------------------------------------------------
$display:

%d or %D   Decimal format

%b or %B   Binary format

%h or %H   Hexadecimal format

%o or %O   Octal format

%c or %C   ASCII character format

%v or %V   Net signal strength

%m or %M   Hierarchical name

%s or %S   As a string

%t or %T   Current time format
```

```
example:
$display("Value a: %o", a) ; // a = 5'b1x0zz ;
value a : XZ
$display("Value a: %o", a) ; // a = 5'bxxxxx ;
value a : xx
reg [8:0] a ; // a = 492 ;
reg [7:0] b ; // b = 205 ;
$display("The decimal value of a is: %d", a) ;
$display("The octal value of a is: %o", a) ;
The decimal value of a is: 492
The octal value of a is: 754
-----------------------------------------------------
blocking and nonblocking assignments:

both are inside sequential block

blocking : =
      changes value immeditley

Non-blocking: <=
      changes only affect at the end of the sequential block
      or after a delay #
      //last value will be assigned

-----------------------------------------------------
Always Blocks:

starts at the begining of simulation time until end of simulation
they are alive in parallel along side other parallel elements such as
continues assignments
always blocks should have some delays or controlled by and event (they
should be controlled to avoid infinite loop)
in the always block everything in sequential

type of always:

always @(signal1 or signal2 or ...) begin //in the parenthizes , they
called sensivity list
...
end

always @(posedge clk)begin
...
end

always @(*)begin //* means any changes to read signals inside the block
                            // like if else readed signals or case
statement or right side of assignment
...
end

-----------------------------------------------------
################################################
########### CLOCK GENERATION ###################
################################################
reg clk = 1'b1;
always @(clk)
clk <= #5 ~clk
-----------------------------------------------------
```

```
use of clock:
always @(posedge clk)
      q <= i
-------------------------------------------------------

#########################################################################
##############   MODULE STRUCTURE IN VERILOG $$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


`timescale 1ns/1ns

module module_name
      #(
            parameters
      )
      (
            port declaration
      )
;


wire and variables declaration



parralell blocks:

continues assignments

module instantion

always @(*) blocks
                                                      // remember always are
executed concurent but the blocks excutes sequential
always @(posedge clock)


test bench (usually in a separated file):
initial blocks



endmodule
-------------------------------------------------------------
################e.g. for a MODULE#####################
            ######################################

`timescale 1ns/1ns

module adder_subtractor
      #(
            parameter nb = 32
      )
      (
            input sub,
            input [nb - 1 : 0] a,
            input [nb - 1 : 0] b,
            input [nb - 1 : 0] s,
            output c,
```

```
            output z,
            output n,
            output v
      );

      wire [nb-1:0] bb = b ^ {nb{sub}}; //combined wire declaration and
assignment

      assign {c,s} = a + bb +sub;

      assign z = s == 0;

      assign n = s[nb - 1];
      assign v = ....(overflown)//too long
      ///and so on....
      // very bad ! instead use always block
      // also v should be wire
endmodule
-------------------------------------------------------------
#############################################################
############# e.g. always @(*) use for v in above example####
#############################################################
always @( * )
begin
      v = 0 //because of avoiding combinational loops
                  // v is reg because it is used in always block
      if( sub == 0)
      begin
            if ( a[nb-1] == b[nb-1])
                  if (s[nb-1] != a[nb-1])
                        v = 1;
      end
      .... //like above
end
-------------------------------------------------------------
combinational logic modeling rules:
use always @(*)
use =           // blocking assignment
make sure all outputs (LHS)are assigned in all condinational branches
(statements)
assign a default value at the begining
do not use any output signals as input
-------------------------------------------------------------
#############################################################################
###
#######REMEMBERRRR!!!! the output for test benches are ALWAYS
wire!!!#######
#############################################################################
###
###############e.g. for testbench##################
#################################################

`timescale 1ns/1ns

module adder_subtractor_TestBench;

parameter num_test = 20;

reg s;
integer i;
```

```
wire signed [7:0] z;
reg signed [7:0] x,y;

adder_subtractor
#(
      .nb(8)
)
uut(
.sub(s),
.a(x),
.b(y),
.s(z),
.o(),
.z(),
.n(),
.v()
);
initial
      for(i = 0 ; i < num_test;i = i + 1)
      begin
            x = $random;
            y = $random;
            z = $random;

            #3; //wait for module to initialize
            if (s)
                  $display("0x%x(%d) - 0x%x(%d) = 0x%x (%d), %0s",
                  x,x,y,y,z,z,!uut.v ? "ok" : "overflown");
            else
                  $display("0x%x(%d) + 0x%x(%d) = 0x%x (%d), %0s",
                  x,x,y,y,z,z,!uut.v ? "ok" : "overflown");

            #7;
      end

endmodule
-------------------------------------------------------
Different between behavorial dataflow and procedural:


    procedural                  dataflow

always @(a or b)                assign q = ~(a | b);
    q = ~(a | b);

// q is reg                      //q is wire
-------------------------------------------------------
reduction operators
assign  q1  = &a; // reduction-and
assign q2 = |b;

examples:
 |(4'b0001) = 1

 ^(4'b0111) = 1

 ~|(2'b11) = 0

 &(2'b1x) = x
-------------------------------------------------------
```

case statement:

Two differences with if-then-else
1. if-then-else conditions can be
   more general. Each if-then-else
   leg can test a different expression

2. case statement performs exact
   matching*, including x and z
   If-then-else always returns false
   when matching x or z (case equality ===)

e.g. 1 :

```
    always @(posedge clk)
    case (state)
    s0: if (a)
        state <= s1;
    else
        state <= s2;
    s1: if (a)
        state <= s2;
    else
        state <= s1;
    s2: state <= s2;
        default: state <= s0;
    endcase
```

e.g. 2 (dont cares):

```
always @(posedge clk)
casez (instr)
    7'b1???????: // arithmetic
    7'b01??????: // load-reg
    7'b00??????: // store-reg
endcase
```

e.g. 3:

```
reg [4:0] thebits;

always @(posedge clk)
case (2'b10)
    thebits[1:0]: // it's on the lsb position
    thebits[2:1]: // it's on the lsb+1 position
    thebits[3:2]: // it's on the lsb+2 position
endcase
```

---

parallel blocks and disabling them:

Give a parallel or sequential block a name

```
initial fork : stimuli
    #50 clk = 0;
    #80 begin
    clk = 1;
    rst = 0;
end
#130 clk = 0;
```

```
#180 rst = 1;
join


  Named blocks can be disabled (terminated)
initial begin : runthis
    forever
    begin
    #10 a = a + 1;
    if (a > 30)
        disable runthis;
    end
end
```

```
################################################
###########STATE MACHINE e.g.#################
################################################

module mealySM
#(
parameter A = 1'b0,
parameter B = 1'b1
)
(
input clock,
input w,
input reset,
output reg z
);

reg CS,NS;

always @(*)

case (CS)
    A: if(w == 0)
        begin
            NS = A;
            z = 0;
        end
    else
        begin
            NS = B;
            z = 0;
        end
    B: if (w == 0)
        begin
            NS = A;
            z = 0;
        end
    else
        begin
            NS = B;
            z = 1;
        end
endcase

always @(posedge clock, negedge reset)
if (reset == 1)
```

```
    CS <= 0;
else
    CS <= NS;

endmodule
-------------------------------------------------------
```