

# Verilog:

**System Tasks,**  
Blocks, Conditional Statement, Loops

# Verilog Module Structure

## Module STARTING

```
module design
(
  input [2:0] data,
  output [2:0] result
);
```

Module name portlist , port declaration  
parameter declarations

```
reg reset_n, clk;
reg done;
reg FF_Q, FF_D;
integer i;
wire [2:0] data_in, data_out;
```

Declaration of wire and  
reg variables

```
alg_module alg_instance (
  .clk (clk ),
  .reset_n (reset_n),
  .d_in (data_in),
  .d_out (data_out)
);
```

Instantiation of  
other module

```
assign data_in = 3'b001 + data;
assign result=data_out ^ data_in;
```

Continuous statements  
or Data flow statement

```
initial begin
  forever #(5) clk=~clk;
end
```

Initial block:  
blocking assignments

always block:  
Non-blocking  
assignments

```
always @(*) begin
  done=0;
  if ( start==1 ) begin
    done=1;
  end else if ( start==0 ) begin
    done=0;
  end else begin
    done=1;
  end

  case(FF_D)
    1'b0: begin
      FF_D=0;
    end
    1'b1: begin
      FF_D=1;
    end
  endcase
end
```

always block:  
Non-blocking  
assignments

```
always @ ( posedge clk or negedge reset_n) begin
  if (reset_n ==0) begin
    FF_Q <= 0 ;
  end else begin
    FF_Q <= FF_D;
  end
end
```

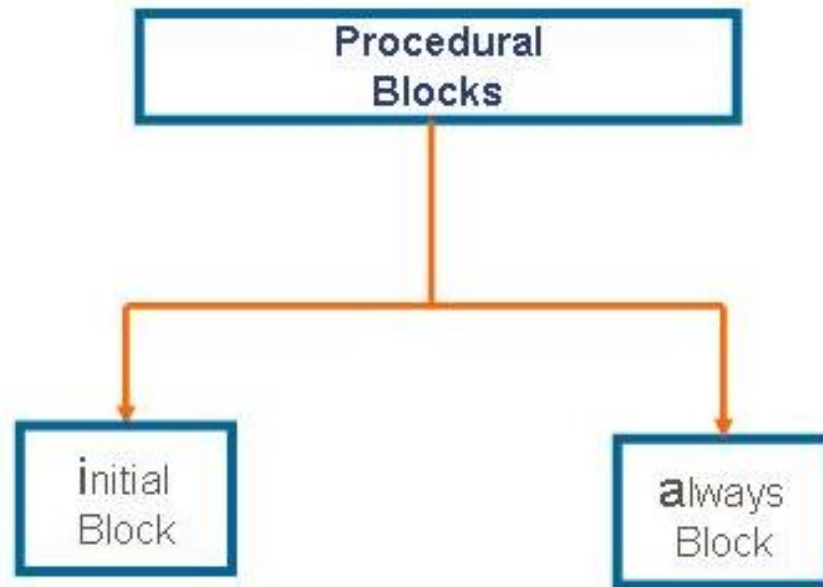
tasks and  
functions

```
task decrypt_task ;
endtask
```

Module ENDING endmodule

# Block Types

Procedural Block consists of two blocks!



# Initial and Always

- ▶ *Multiple statements per block*

- Procedural assignments

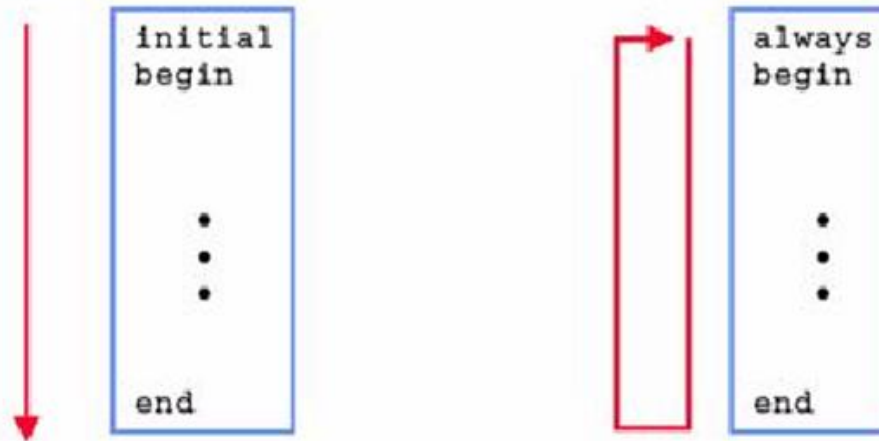
- Timing control

- ▶ *Initial blocks execute once*

- at  $t = 0$

- ▶ *Always blocks execute continuously*

- at  $t = 0$  and repeatedly thereafter



# always and initial block syntax

- Initial block syntax:

```
Initial begin
    statement1;
    statement2;
    ....
End
```

OR

```
Initial statement;
```

- always block syntax

```
always @ ( sensitivity list) begin
    ....
end
```

OR

```
always @ ( * )
begin .... end
```

# Initial Block

- ▶ This block starts with *initial* keyword
- ▶ This is non synthesizable
- ▶ Non RTL
- ▶ This block is used only in stimulus
- ▶ All initial blocks execute concurrently in arbitrary order
- ▶ They execute until they come to a #delay operator
- ▶ Then they suspend, putting themselves in the event list delay time units in the future
- ▶ At delay units, they resume executing where they left off

# System Tasks and Functions

- System Task and Functions start with \$
- Common tasks and functions
  - Display the value of variables
    - **\$monitor** ("text\_with\_format\_specifiers", *signal*, *signal*, ... );
    - **\$display** ("text\_with\_format\_specifiers", *signal*, *signal*, ... );
  - Simulation control : Stop (i.e. suspend) , finish (i.e. exit) simulation
    - **\$finish** (Finishes a simulation and exits the simulation process).
    - **\$stop** (Halts a simulation and enters an interactive debug mode).

# Display the value of variables

- **\$monitor** ("text\_with\_format\_specifiers", *signal*, *signal*, ... );
  - Continuously monitors the signals listed, and prints the formatted message whenever one of the signals changes. A newline is automatically added to the text printed.
- **\$display** ("text\_with\_format\_specifiers", *signal*, *signal*, ... );
  - Prints the formatted message once when the statement is executed during simulation. A newline is automatically added to the text printed.

Text Formatting Codes			
%b	binary values	%s	character strings
%o	octal values	%m	hierarchical names
%d	decimal values	\t	print a tab
%h	hex values	\n	print a <u>newline</u>
%e	real values-exponential	\"	print a quote
%f	real values-decimal	\\	print a backslash
%t	formatted time values	%%	print a percent



# \$display Example

```
module helloworld;
```

```
  reg [7:0] a;
```

```
  initial begin
```

```
    #10 a = 30;
```

```
    #20 a = 40;
```

```
  end;
```

```
  initial begin
```

```
    $display("a = ", a);
```

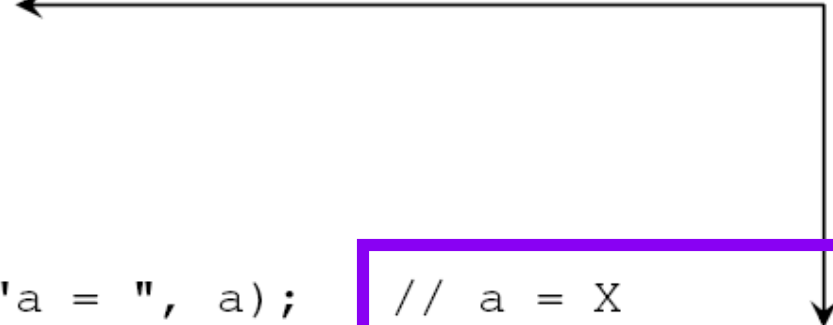
```
  #10 $display("a = ", a);
```

```
  #10 $display("a = ", a);
```

```
  #10 $display("a = ", a);
```

```
  end
```

```
endmodule
```



```
// a = X  
// a = 30 indeterminate  
// a = 30  
// a = 40;
```

Output of \$display task

# More \$display Examples

```
module helloworld;

reg [7:0] a;

initial begin
    a = 43;
    $display("a = %b", a);
    $display("a = %x", a);
    $display("a = %c", a);
    $display("a = %o", a);
end

endmodule
```

a = 00101011  
a = 2b  
a = +  
a = 053

```
module helloworld;

reg [7:0] a, b;

initial begin
    a = 43;
    b = 68;
    $display("a = %b and b = %d", a, b);
end

endmodule

a = 00101011 and b = 68
```

```
module helloworld;

initial begin
    #10 $display("Module: %m");
    $display("Time: %d", $time);
    $display("Library: %l");
end

endmodule

Module: helloworld
Time: 10
Library: work.helloworld
```

# \$monitor Example

```
module helloworld;

reg a;

initial begin
    a = 0;
    # 35 a = 1'bx;
end

always #10 a = ~a;

initial #15 $display("%d d1", $time, a);

always @(a) $display("%d d2", $time, a);

initial $monitor("%d d3", $time, a);

endmodule
```

\$time: returns simulation time

0 d2 0  
0 d3 0  
10 d2 1  
10 d3 1  
15 d1 1  
20 d2 0  
20 d3 0  
30 d2 1  
30 d3 1  
35 d2 x  
35 d3 x

# Initial Block Example

initial Block starts at time 0,  
executes only once during a  
simulation

Multiple statements must be grouped  
using the keywords begin and end.  
Incase of a single behavioral  
statement, grouping is not necessary.

Here we have three initial blocks  
where each block starts to  
execute in parallel at time 0.  
Each Block finishes its execution  
independent of each other.

time	statement executed
0	c=1'b0;
5	a=1'b1;
10	x=1'b0;
30	b=1'b0;
35	y=1'b1;
50	\$finish;

```
module stimulus;  
  reg x,y, a, b, c;  
  initial  
  begin  
      c=1'b0;  
      #5 a=1'b1;  
      #25 b=1'b0;  
  end  
  initial  
  begin  
      #10 x=1'b0;  
      #25 y=1'b1;  
  end  
  initial  
      #50 $finish;  
endmodule
```

# always Block

- ▶ This block starts with *always* keyword.
- ▶ This block is more like H/W.
- ▶ The always block can be viewed as continuously repeated activity in a digital circuit starting from power on.

always Block starts at time 0.

Clock is initialized inside a separate initial statement. If we initialize the clock inside the always block, clock will be initialized every time the always block is entered.

always Block executes the statement `clock=~clock` every 10 time units.

```
module clock_gen
    reg _clk;

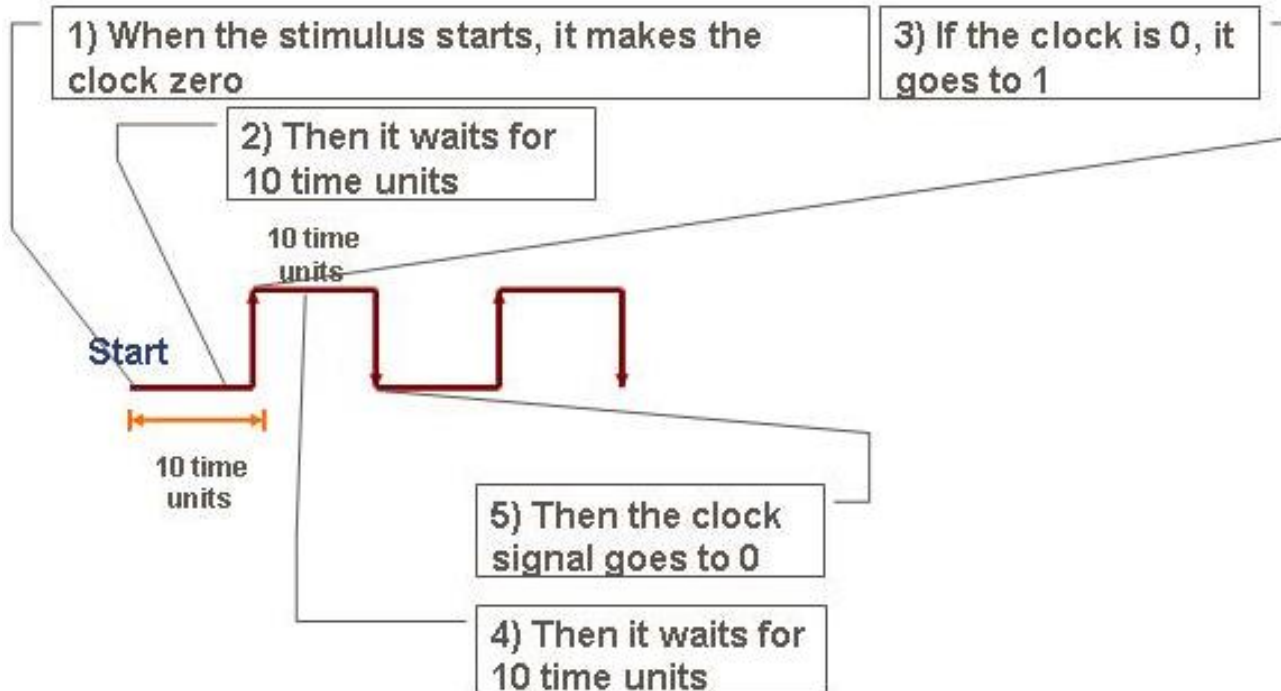
    initial
        clk = 1'b0;

    always @ (clk)
        #10 clk = ~ clk ;

    initial begin
        #1000 $finish;
    end

endmodule
```

# always Block



```
module clock_gen
  reg _clk;

  initial
    clk = 1'b0;

  always @ (clk)
    #10 clk = ~ clk ;

  initial begin
    #1000 $finish;
  end

endmodule
```

# More always Block Examples

Always Block Examples	Notes
<pre>always @(a or b or ci) begin     sum = a + b + ci; end</pre>	always block executes statements repeatedly.
<pre>always @(posedge clk)     q &lt;= data;</pre>	always block executes when clock changes from 0 to 1

# Conditional Statements

- If statement syntax:
  - `if` (*expression*) *statement or statement\_group*
  - `if` (*expression*) *statement or statement\_group*  
`else` *statement or statement\_group*

```
reg [7:0] a, b, max;
```

```
always @(posedge clk) begin
```

```
    if (a > b) begin
```

```
        max = a;
```

```
        ... other statement ..
```

```
    end
```

```
end
```

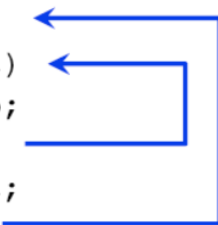
Use begin .. end  
to include  
multiple statements



# Conditional Statements: Nested If

```
reg [7:0] a, b, c, max;

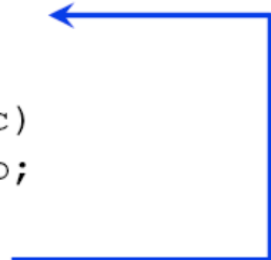
always @(posedge clk) begin
    if (b > a)
        if (b > c)
            max = b;
        else
            max = c;
    else
        max = a;
end
```



Note that indentation does not  
imply association.  
Verilog is context-free.

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
    if (b > a)
        begin
            if (b > c)
                max = b;
            end
        else
            max = a;
    end
```



# Conditional Statements: Multi-way If

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
    if (a < 10) begin
        // ... when a < 10
    end
    else if (a < 20) begin
        // ... when a >= 10 and a < 20
    end
    else if (a < 30) begin
        // .. when a >= 20 and a < 30
    end
    else begin
        // .. when a >= 30
    end
end
```

# Conditional Statements: case statement

- Case syntax

```
case (net_or_register_or_literal)  
  case_match1: statement or statement_group  
  case_match2,  
  case_match3: statement or statement_group  
default: statement or statement_group  
endcase
```

# Loop Statements

- ▶ repeat:  
repeat a fixed number of times
- ▶ while:  
conditionally repeat a number of times
- ▶ for:  
initialize variable;  
test loop condition;  
run body;  
loop counter update

# Forever

## Syntax:

`forever` statement or statement\_group

```
reg clk;
```

```
Initial begin
```

```
    clk = 0;
```

```
    forever #10 clk = ~ clk;
```

```
end
```

# Repeat

## Syntax:

```
repeat (number)  
  
    statement or  
    statement_group
```

```
module my_design;  
    initial begin  
        repeat(4) begin  
            $display("This is a new iteration ...");  
        end  
    end  
endmodule
```

## Simulation output:

```
This is a new iteration ...  
This is a new iteration ...  
This is a new iteration ...  
This is a new iteration ...
```

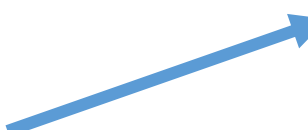
# While Statement

Syntax:

**while** (*expression*) statement or  
statement\_group

```
module countones(num, in);  
    input  [7:0] in;  
    output [7:0] num;  
    reg     [7:0] num;  
    reg     [7:0] wrk;  
  
    initial begin  
        num = 0;  
        wrk = in;  
        while (wrk) begin  
            if (wrk[0])  
                num = num + 1;  
            wrk = wrk >> 1;  
        end  
    end  
endmodule
```

Count number of 1's in  
register **in**



# For Statement

## Sntax:

```
for (initial_assignment,  
    expression;  
    step_assignment)  
statement or statement_group
```

```
module countones(num, in);  
    input  [7:0] in;  
    output [7:0] num;  
    reg     [7:0] num, a;  
    reg     [7:0] wrk;  
  
    initial begin  
        num = 0;  
        wrk = in;  
        for (a=0; a < 8; a = a + 1)  
            if (wrk[a])  
                num = num + 1;  
    end  
endmodule
```



# Parallel Statements

Syntax:

fork

statement1;

statement2;

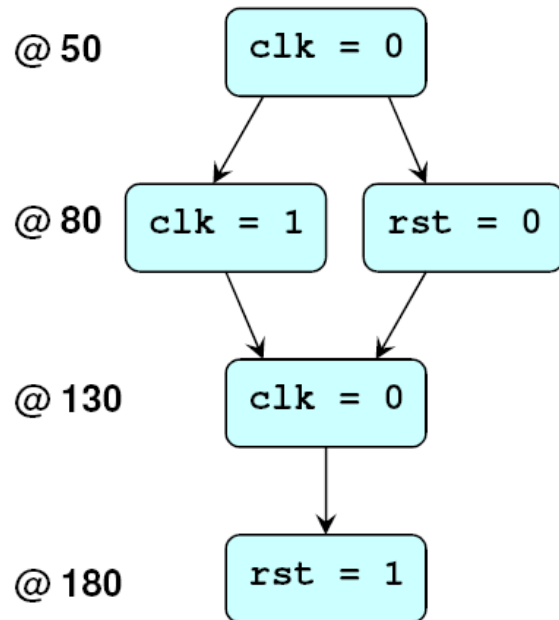
...

statementN;

join

**All statements  
start at the  
same time.**

# Parallel Statements



```
initial begin
    #50 clk = 0;
    #30 clk = 1;
        rst = 0;
    #50 clk = 0;
    #50 rst = 1;
end
```

```
initial fork
    #50 clk = 0;
    #80 clk = 1;
    #80 rst = 0;
    #130 clk = 0;
    #180 rst = 1;
join
```

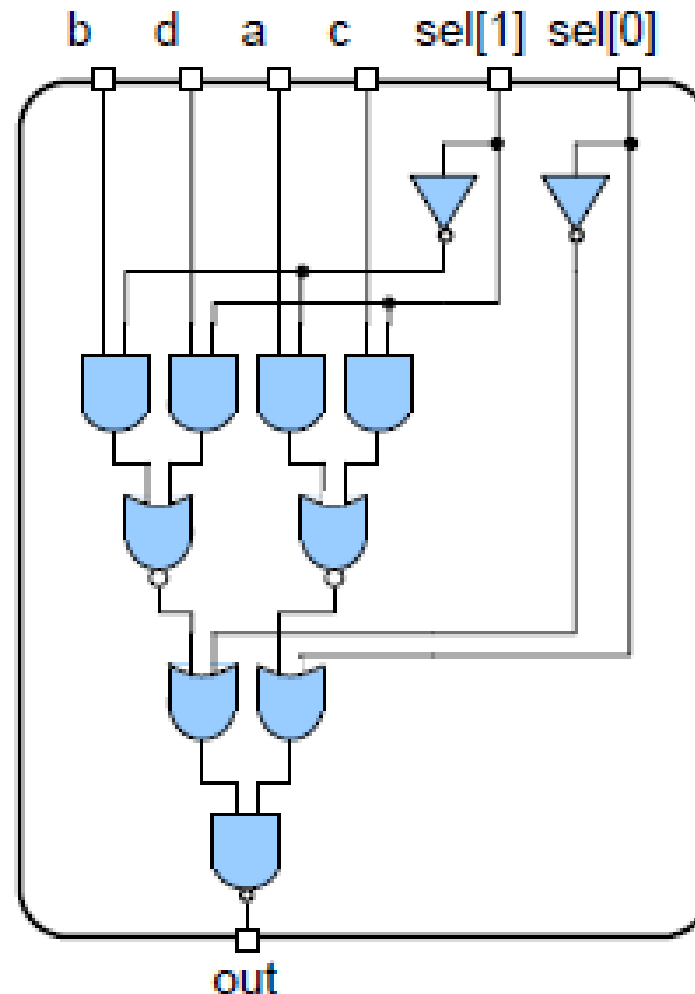
`begin/end` and `fork/join` blocks  
have same effect

# Mux4 Modeling

- ▶ Will model the mux4 in various modeling styles:
  - ▶ Structural
  - ▶ Behavioral Continuous Assignment
  - ▶ Behavioral Procedural using if statement
  - ▶ Behavioral Procedural using case statement

# Structural to Behavioral

```
module mux4( input  a, b, c, d, input [1:0] sel, output out );  
  
    wire [1:0] sel_b;  
    not not0( sel_b[0], sel[0] );  
    not not1( sel_b[1], sel[1] );  
  
    wire n0, n1, n2, n3;  
    and and0( n0, c, sel[1] );  
    and and1( n1, a, sel_b[1] );  
    and and2( n2, d, sel[1] );  
    and and3( n3, b, sel_b[1] );  
  
    wire x0, x1;  
    nor nor0( x0, n0, n1 );  
    nor nor1( x1, n2, n3 );  
  
    wire y0, y1;  
    or or0( y0, x0, sel[0] );  
    or or1( y1, x1, sel_b[0] );  
    nand nand0( out, y0, y1 );  
  
endmodule
```



## Mux 4 Behavioral (Cont. assign)

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    wire out, t0, t1;
    assign t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    assign t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

**The order of these continuous  
assignment statements does not matter.  
They essentially happen in parallel!**

```
// Four input multiplexer
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    assign out = ( sel == 0 ) ? a :
                 ( sel == 1 ) ? b :
                 ( sel == 2 ) ? c :
                 ( sel == 3 ) ? d : 1'bx;

endmodule
```

**If input is undefined we  
want to propagate that  
information.**

**1'bx;**

# Mux4 Example

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out;

    always @( * )
    begin
        if ( sel == 2'd0 )
            out = a;
        else if ( sel == 2'd1 )
            out = b;
        else if ( sel == 2'd2 )
            out = c;
        else if ( sel == 2'd3 )
            out = d;
        else
            out = 1'bx;
    end

endmodule
```

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out;

    always @( * )
    begin
        case ( sel )
            2'd0 : out = a;
            2'd1 : out = b;
            2'd2 : out = c;
            2'd3 : out = d;
            default : out = 1'bx;
        endcase
    end

endmodule
```