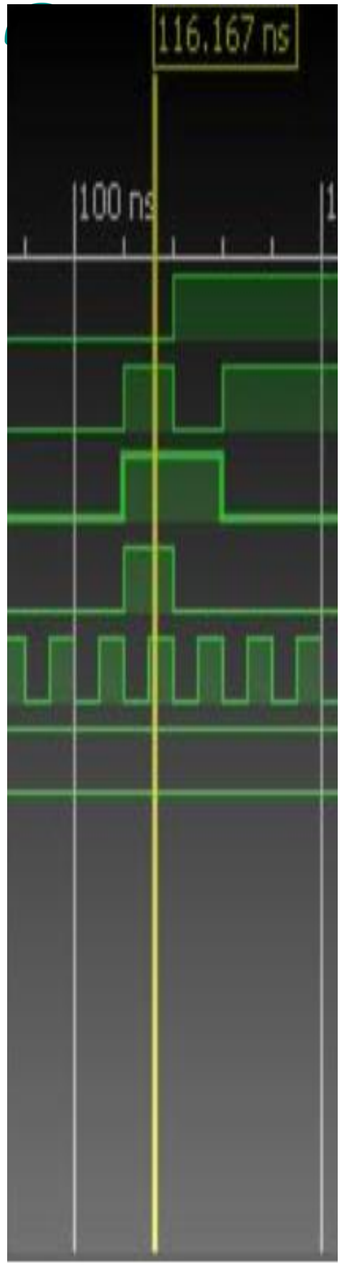
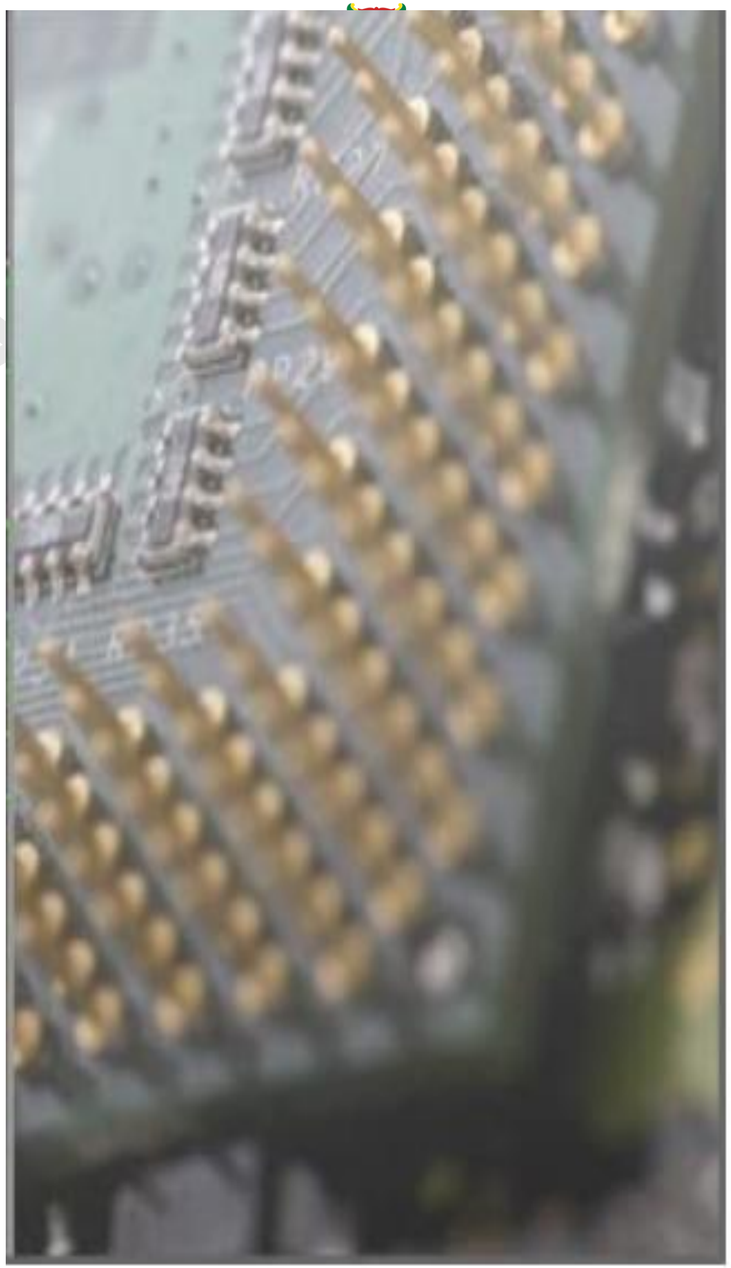
# Verilog II

MC. Martin González Pérez

# Table of contents



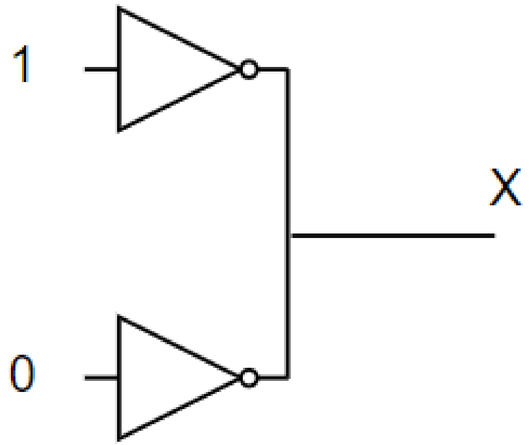

# Data types



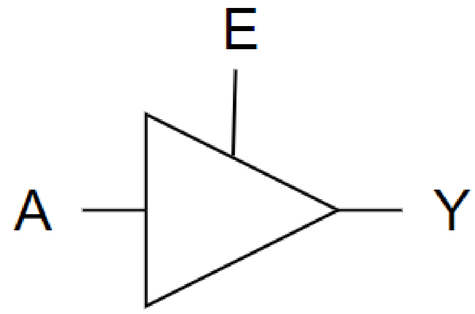
# 4 States

Verilog represent values in 4 possible states:
**0:** Zero, Low, False, Ground.
**1:** One, High, True, VDD.
**Z:** High impedance, Undriven, Unconected, Drive disabled (tri-state).
**X:** Uninitialized, Unknown.






| E | A1 | Y |
|---|---|---|
| 0 | 0 | **Z** |
| 0 | 1 | **Z** |
| 1 | 0 | **0** |
| 1 | 1 | **1** |



# Data types

**Nets:** Represent **physical connection**. Nets can be driven everywhere **outside procedures**.
Example: wire, tri, wand, triand, wor, trior.

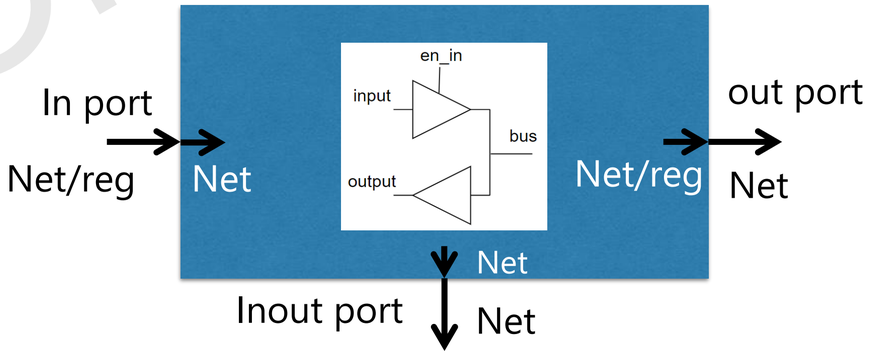
**Variables:** Represent abstract **storage elements**. Variables can only be driven **inside procedures.**
Example: integer, real, reg, time, realtime. Only reg is synthesizable**.**

**Parameters:** Run-time constants.
Example: parameter, localparam.

# Ports data types

- In Verilog, ports are implicitly declared as wire nets unless you explicitly declare it otherwise.
- Modules inputs are always nets.
- Modules outputs are variables if driven by a procedural block; otherwise, they are nets.
- Connections to inputs ports are variables if  are driven by a procedural block; otherwise, they are nets.
- Connections to output ports are always nets.
- Connections to bidirectional ports are always nets.





# Declaring vectors

A vector is a net or reg with a range specification.
For declaring vector, you must specify the range (ascending or descending) when declaring the variable or net as follow:

[data_type] [msb_constant : lsb_constant] [signal_name];

Example:

```
reg [3:0] flop1;
reg [0:3] flop2;
```

# Different bit widths

It is possible to do assignments with different width vectors

Unsigned source shorter than target fill with zeros msb.
Unsigned source shorter than target fill with sign-value msb.
If source is wider than targer truncates values from msb.

```
reg [7:0] a;
reg [3:0] b;
…
```

| a = b; | b = a; |
|---|---|
| a[0]<- b[0] | b[0]<- a[0] |
| a[1]<- b[1] | b[1]<- a[1] |
| a[2]<- b[2] | b[2]<- a[2] |
| a[3]<- b[3] | b[3]<- a[3] |
| a[4]<- 0 | |
| a[5]<- 0 | |
| a[6]<- 0 | |
| a[7]<- 0 | |

## Concatenation

| a = {b,b}; | a = {4'b1010,b}; |
|---|---|
| a[0]<- b[0] | a[0]<- b[0] |
| a[1]<- b[1] | a[1]<- b[1] |
| a[2]<- b[2] | a[2]<- b[2] |
| a[3]<- b[3] | a[3]<- b[3] |
| a[4]<- b[0] | a[4]<- 0 |
| a[5]<- b[1] | a[5]<- 1 |
| a[6]<- b[2] | a[6]<- 0 |
| a[7]<- b[3] | a[7]<- 1 |

# Literal values

You can specify litel values as:
[size] ‘ [base] [value]

**Size:** positive decimal value that specifie the size of the bus, if unsize is 32 bits.
**Base:** indicate the radix: binary(b/B), hexadecimal(h/H) or decimal(d/D).
**Value:** legal digits for base. Can include “_” if not 1st character. Can include Z or X digits.

*Example:*

```
reg [3:0] flop1;
…
flop1 = 4'b1101;     //1101
flop1 = 4'd12;       //1100
flop1 = 4'd16;       //0000
flop1 = 4'h2f;       //1111
```

# Bit select/Range select

It is possible to select a bit or range of bits in a vector as is shown below.

```
reg [7:0] flop;          //vector
wire msb;
wire lsb;
wire [3:0]nible1;
wire [3:0]nible2;
…
flop = 8'b1100_1010;
//Bit select
msb = flop[7];           // 1
lsb = flop[0];           // 0
//Range select
nible1 = flop[3:0];      //1010
nible2 = flop[7:4];      //1100
```

# Multiple drivers on nets

Nets can solve multiple drivers.

```
assign y = a;
assign y = b;
```

wire/tri

| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 0 | X | 0 | X |
| 1 | X | 1 | 1 | X |
| Z | 0 | 1 | Z | X |
| X | X | X | X | X |

wand/triand

| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | X |
| Z | 0 | 1 | Z | X |
| X | 0 | X | X | X |

wor/trior

| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |
| Z | 0 | 1 | Z | X |
| X | X | 1 | X | X |

# Arrays

Verilog supports arrays.

```
reg [7:0] mem [0:255];        // 256 bytes
//access to a bit
bit_acces = mem[100][0];      //lsb from byte 100
```

Verilog 2001 support multi-dimensional arrays.

```
reg [7:0] mem [0:255][0:9];
```

# Module parameters

We can use parameter to define constants in modules.

```
module bw_and #(parameter WIDTH = 8) // parameters
(input [WIDTH-1:0] a, input [WIDTH-1:0] b, output out); //ports
    assign out = a & b;
endmodule
```

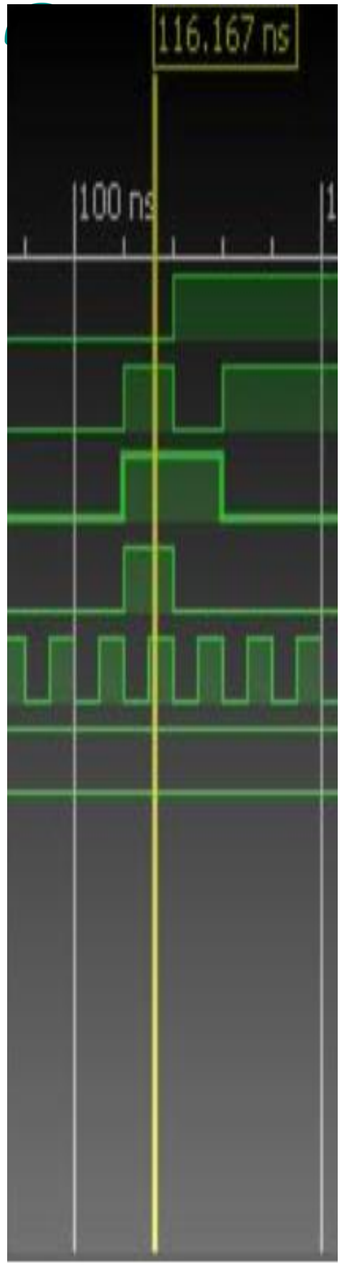
Also, we can change the parameter value in each instance.

```
module top (ports);
…
    bw_and #(.WIDTH(4)) bw_and_inst0 (.a(a0), .b(b0), .out(out0));
    bw_and #(.WIDTH(4)) bw_and_inst1 (.a(a1), .b(b1), .out(out1));
…
endmodule
```

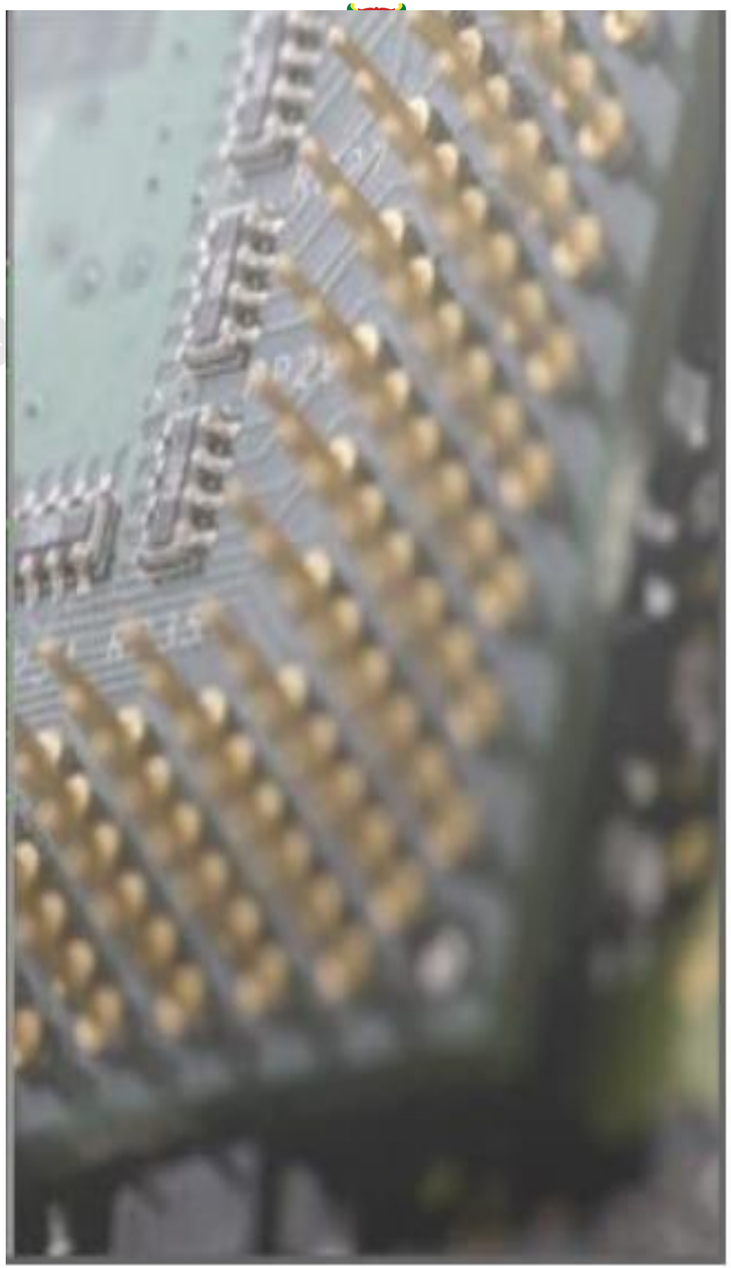
# Local parameters

Inside modules use localparam construct to define constant that does not change.

```
module calc_exp #(parameter WIDTH = 8)
(input [WIDTH-1:0] a, input [WIDTH-1:0] b, output input [WIDTH-1:0] exp);
    localparam bias = 127;
    assign exp = a+b-bias;
endmodule
```




# Operators



# Verilog operators

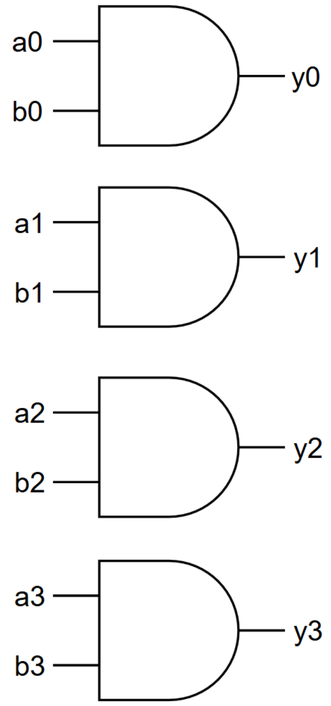
| Category | Symbols |
|---|---|
| Bit-wise | ~, &, \|, ^, ~^ |
| Reduction | &, ~&, \|, ~\|, ^, ~^ |
| Logical | !, &&, \|\| |
| Arithmetic | **, *, /, %, +, - |
| Shift | <<, >>, <<<, >>> |
| Relational | <, >, <=, >= |
| Equality | ==, !=, ===, ~== |
| Conditional | ? : |
| Concatenation | {} |
| Replicarion | {{}} |

# Bit-wise operators

| Operation | Operator |
|---|---|
| not | ~ |
| and | & |
| or | \| |
| xor | ^ |
| xnor | ~^, ^~ |

- Bit-wise operators operate on vectors.
- Operation are performed bit by bit.

```
module bitwise;
    reg [3:0] opa, opb;
    wire [3:0] res_and, res_or, res_not;

    assign res_and = opa & opb;
    assign res_or = opa | opb;
    assign res_not = ~opa;

    initial begin
        opa = 4'b0101;
        opb = 4'b0011;
        #10;
        $display ("%b", res_and);//0001
        $display ("%b", res_or); //0111
        $display ("%b", res_not);//1010
    end
endmodule
```

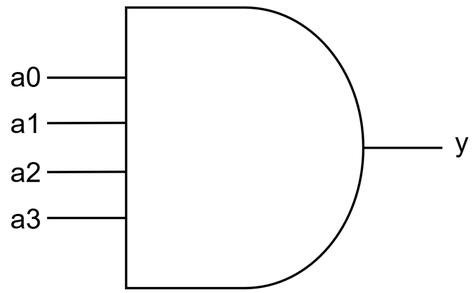


# Reduction operators

| Operation | Operator |
|---|---|
| and | & |
| or | \| |
| nand | ~& |
| nor | ~\| |
| xor | ^ |
| xnor | ~^, ^~ |

- Reduction operators perform a bit-wise operation with all the bits of a single operand.
- The result is always 1-bit width.

```
module reduction;
    reg [3:0] opa, opb;
    wire res_and, res_or;

    assign res_and = &opa;
    assign res_or = |opa;

    initial begin
        opa = 4'b0101;
        opb = 4'b0011;
        #10;
        $display ("%b", res_and);//0
        $display ("%b", res_or); //1
    end
endmodule
```



# Logical operators

| Operation | Operator |
|---|---|
| not | ! |
| and | && |
| or | \|\| |

- Logical operators interpret their operand as 0 if all bits are 0, 1 if any bit is 1 or unknow (x) if any bit is z or x and no bit is 1.
- The operands are vectors, but the result is 1-bit.

```
module logical;
    localparam [3:0] A = 4'b0011;    //1
    localparam [3:0] B = 4'b10xz;    //1
    localparam [3:0] X = 4'b0xz0;    //X
    localparam [3:0] ZERO = 4'b0000;//0
    reg res;

    initial begin
        res = !A;         //0
        $display ("%b", res);
        res = A&&B;       //1
        $display ("%b", res);
        res = A&&ZERO;    //0
        $display ("%b", res);
        res = B||ZERO;    //1
        $display ("%b", res);
        res = X||ZERO;    //X
        $display ("%b", res);
    end
endmodule
```

# Signed vectors

We can use signed keyword for treat literal, nets and regs as a signed values.

```
8'shFA
reg signed [7:0] data_reg = -4; //11111100
wire signed [7:0] data_net = 5; //00000101
```

Signed values are represented as 2 complement.

# Arithmetic operators

| Operation | Operator |
|---|---|
| Add | + |
| Subtract | - |
| Multiply | * |
| Divide | / |
| Modulus | % |
| Exponential power | ** |

```
module arithmetic;
    reg [7:0] opa, opb;
    reg [7:0]val;

    initial begin
        opa = 8'd7;
        opb = 8'd4;
        val = opa + opb; //11
        val = opa - opb; //3
        val = opa * opb; //28
    end
endmodule
```

# Shift operators

| Operation | Operator |
|---|---|
| Logical shif | <<, >> |
| Arithmetic shift | <<<, >>> |

- Logical shift fills extra bits with 0 and ignores operand sign.
- Left arithmetic shift operates like logical shift.
- Right arithmetic shift preserves the operand sign in the result.

```
module shifters;
    reg [3:0] opa = 4'b1010;
    reg signed [3:0] opb = 4'b101
    reg [3:0] res;
    initial begin
        res = opa << 1;  //0100
        $display ("%b", res);
        res = opa >> 1;  //0101
        $display ("%b", res);
        res = opa <<< 1; //0100
        $display ("%b", res);
        res = opb >>> 1; //1101
        $display ("%b", res);
        res = opa >>> 1;  //0101
        $display ("%b", res);
    end
endmodule
```

# Relational operators

| Operation | Operator |
|---|---|
| Less than | < |
| Greather than | > |
| Less tan or equal to | <= |
| Greather than or equal to | >= |

Result is:

- 1'b0 if the relation is false.
- 1'b1 if the relation is true.
- 1'bx if either operand contains any z or x bits.

```
module relational;
reg [3:0] opa, opb, opc;
reg res;

initial begin
    opa = 4'd5;
    opb = 4'd2;
    opc = 4'b01x0;

    res = opa < opb; //0
    res = opa > opb; //1
    res = opa <= opc;//x
end
endmodule
```

# Equality

| Operation | Operator |
|---|---|
| Logical equality | == |
| Logical inequality | != |
| Case equality | === |
| Case inequality | !== |

Result is:

- For logial equalities, the result is 1'b0, 1'b1 or 1'bx. Can't when one operator is x, the result is x.
- For case equalities, the result is 1'b0 or 1'b1. Can compare x values.

Logical equality

| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 1 | 0 | X | X |
| 1 | 0 | 1 | X | X |
| Z | X | X | X | X |
| X | X | X | X | X |

Case equality

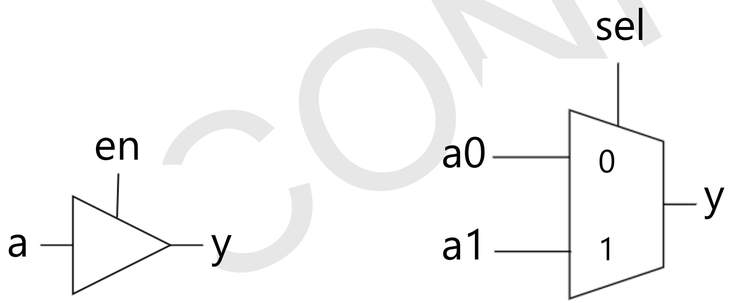
| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| Z | 0 | 0 | 1 | 0 |
| X | 0 | 0 | 0 | 1 |

```
module equalities;
    reg [3:0]a = 4'b0111;
    reg [3:0]b = 4'b1010;
    reg [3:0]c = 4'b0x01;
    reg [3:0]val;

    initial
    begin
        val = a == b;   // 0
        val = a == c;   // x
        val = a === b;  // 0
        val = a !== c;  // 1
        val = b === c;  // 0
    end
endmodule
```

# Conditional operator

| Operation | Operator |
|---|---|
| Conditional | ?: |

This operator works like a mux or tristate.




```
module tri_buff(input a,en, output y);
    assign y = en ? a : 1'bZ;
endmodule

module mux21(input a0,a1,sel, output y);
    assign y = sel ? a0 : a1;
endmodule
```

# Concatenation

| Operation | Operator |
|---|---|
| Concatenation | {} |

Concatenation allow join different bits or range of bits in a single bus.

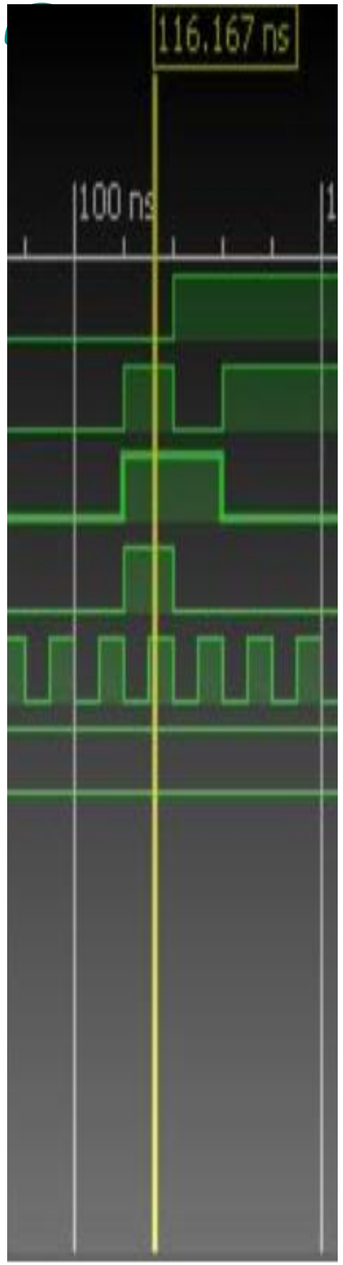
```
module concat;
    reg [7:0] rega, regb;
    reg [3:0] nibc, nibd;
    reg [7:0] val;

    initial
    begin
        rega = 8'b01111000;
        regb = 8'b10101010;
        nibc = 4'b0011;
        nibd = 4'b0001;
        val = {nibc,nibd};                  //00110001
        val = {2'b00,rega[7:4],2'b11};      //00011111
        val = {regb[3:0],nibd};             //10100001
    end
endmodule
```

# Replication

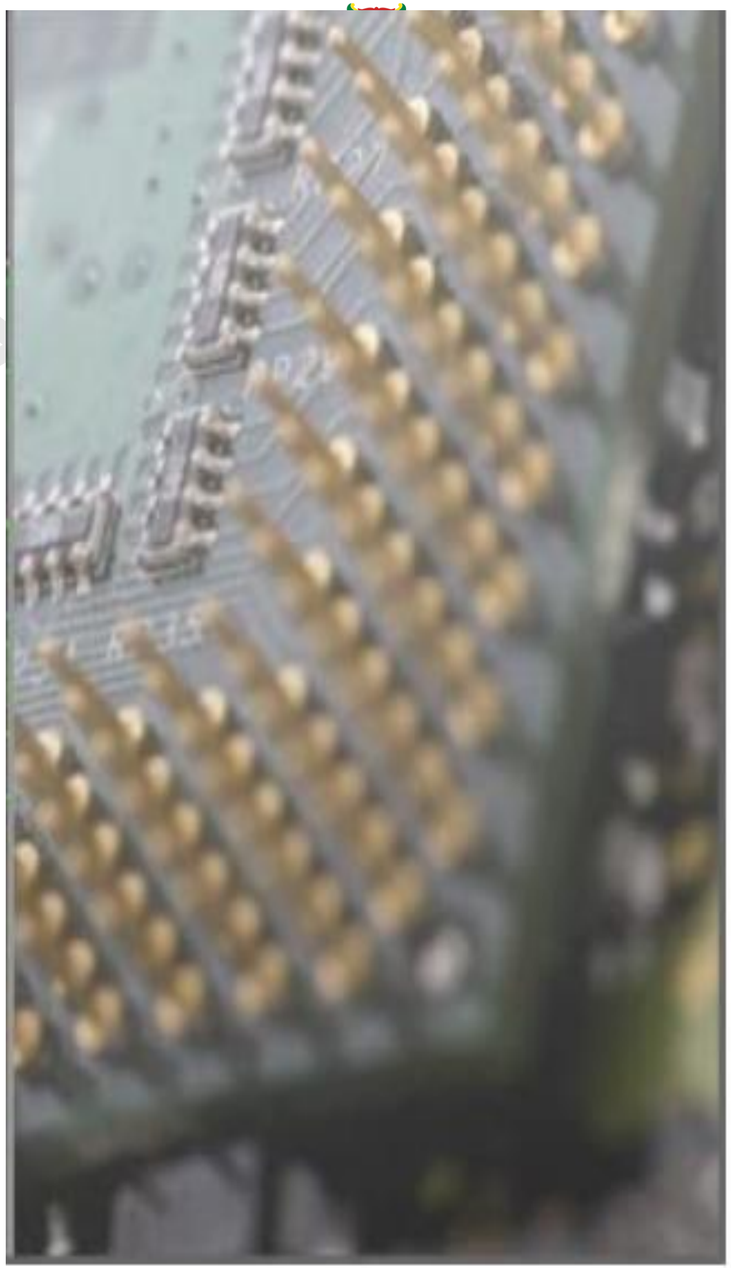
| Operation | Operator |
|---|---|
| Replication operator | {{}} |

Reproduces a concatenation set number of times. It consists of two parameters: The expression (expr) to be repeated and a constant (cons) that indicates how many times the expression will be repeated.
Syntax:
{cons{expr}}.

```
module replication;
    reg [7:0]byte_a = 8'h74;
    reg [7:0]byte_b = 8'hf0;
    reg [31:0] word;
    parameter BYTE_WIDTH = 8;
    initial
    begin
        word = {4{byte_b}}; //32'hf0f0f0f0
        word = {{BYTE_WIDTH{1'b0}},byte_a,byte_b,{BYTE_WIDTH{1'b0}}}; //32'h0074f000
    end
endmodule
```




# Timing control



# Timing control

- **@:** event control (wait for simulation event).
- **wait:** level sensitive event control (conditionally wait for events).
- **#:** delay control.

# @event control

Use @ event control for edge sensitive (**transitions**) timing control.

```
always @(d)
    q = d ;
```

```
always @(posedge clk)
    a <= a + 1;
```

# “wait” event control

Use wait event control for **level-sensitive** timing control.

```
always @(d) begin
    wait(en);
    q = d ;
end
```

# "#" delay control

Use # for simple delay.

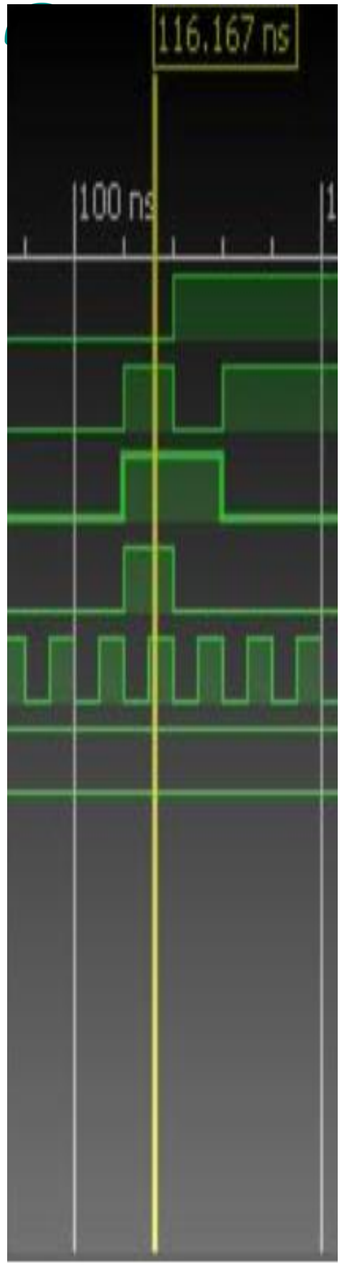
```
always @(d) begin
    wait(en);
    #3 q = d ;
end
```

# Timescale

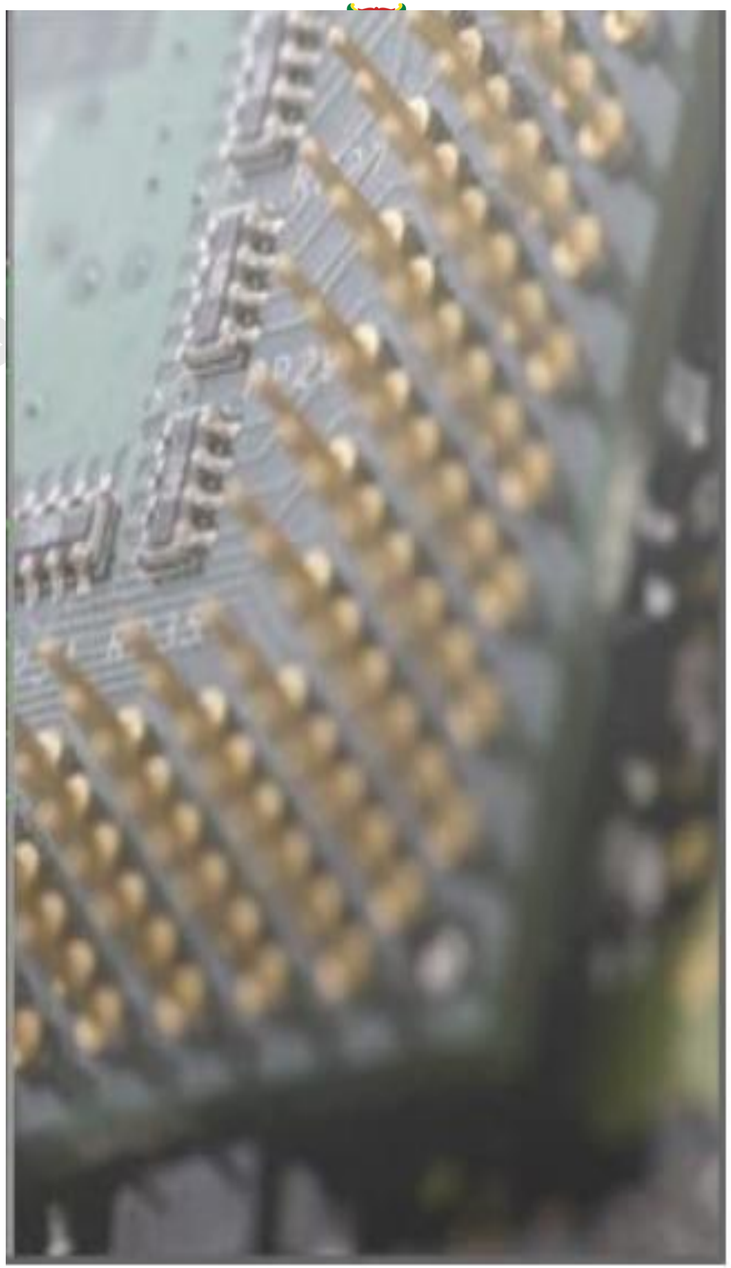
Timescale is a compiler directive that specifies the unit and precision of simulation time.

**`timescale unit/precision**

```
`timescale 1ns/10ps
always @(d) begin
    wait(en);
    #3 q = d ; //3ns delay
end
```




# Continuous assignments

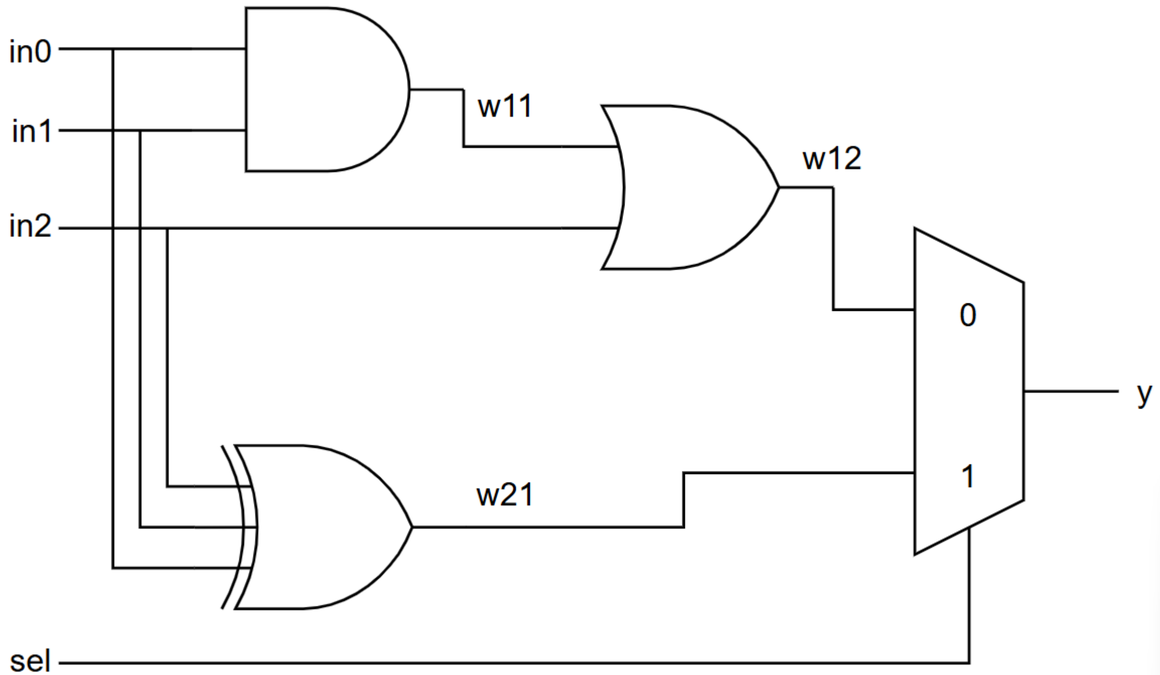


# Continuous assignment

A continuous assignment is a construct used to **drive values onto nets**, such as wire, using the **assign** keyword. It **models combinational logic** by **continuously evaluating the right-hand side expression and updating the net whenever any of its input signals change**. Continuous assignments operate **concurrently**, reflecting the parallel nature of hardware, and are typically **used to describe simple logic functions, signal connections, or constant value assignments** without requiring procedural blocks like always.

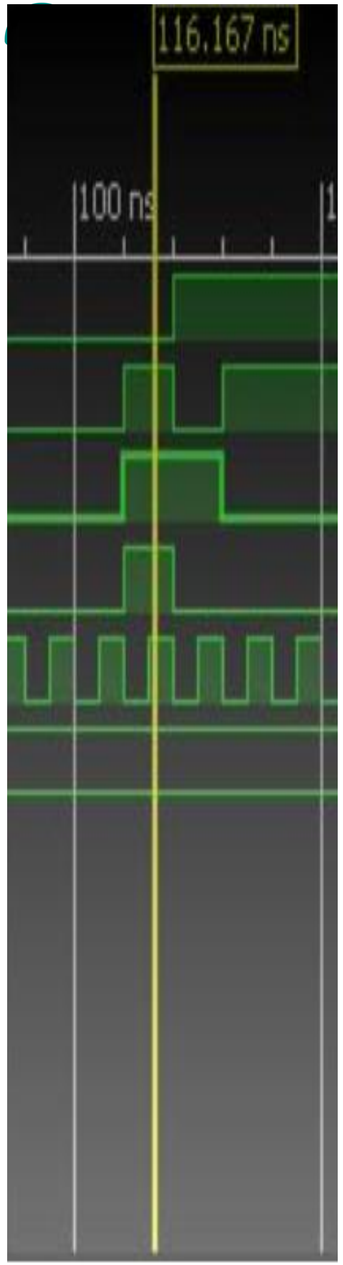
# Continuous assignment

```
module prueba(input [2:0]in, input sel, output y);
    wire w11;
    wire w12;
    wire w21;

    assign w11 = in[0] && in[1];
    assign w12 = w11 || in[2];
    assign w21 = ^in;
    assign y = sel ? w21 : w12;
endmodule
```

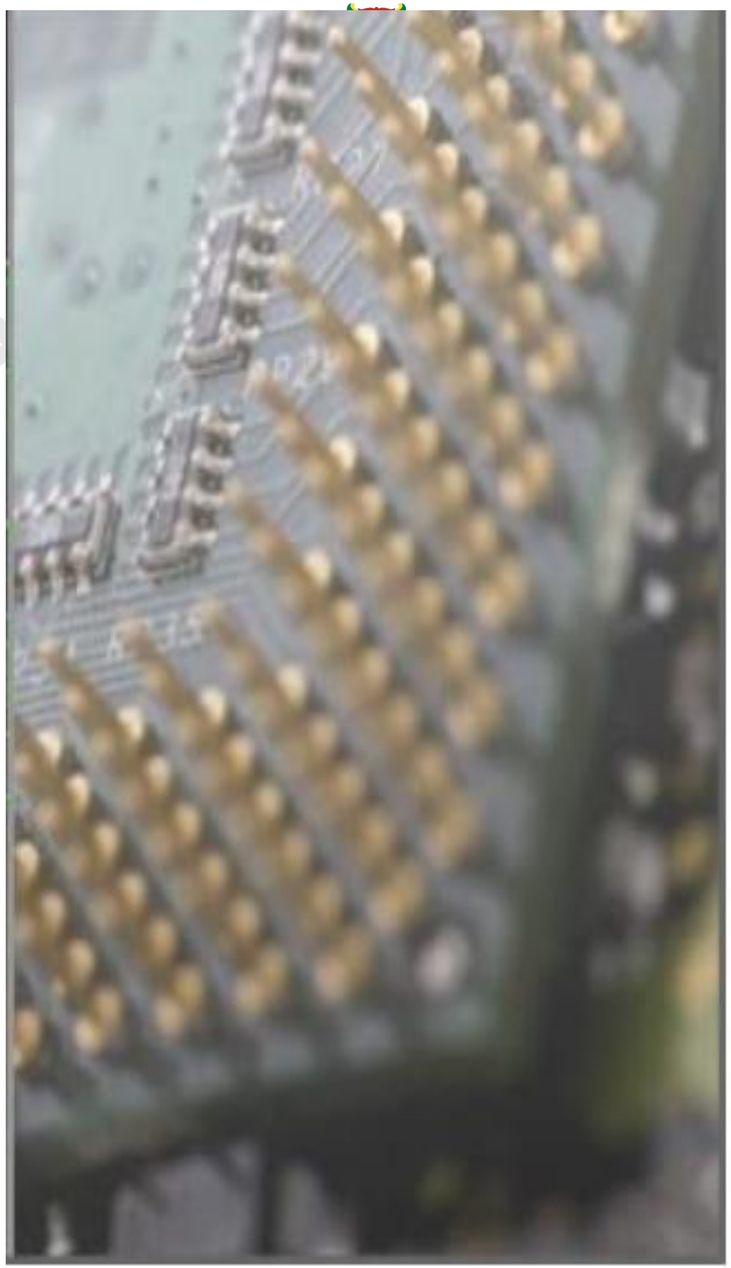


# LAB-1A: BCD to 7-segment display decoder

Design a BCD to 7-seg display decoder **using continuous assignments**.



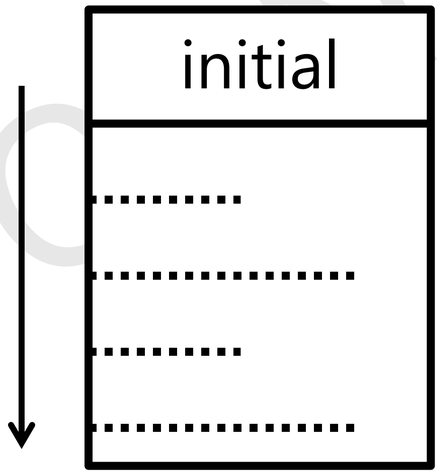

# Procedural assignments

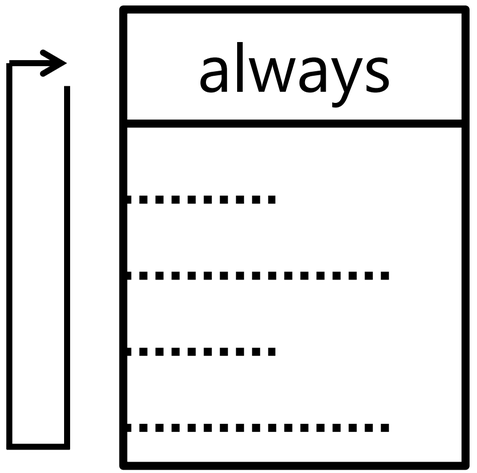


# Procedural blocks

Verilog works through events and processes. **Processes trigger events** (a change in a variable can be an event) and **events trigger processes**. An example of this is continuous assignment, which **updates** the value on the left-hand side whenever one of the operands **changes**. The most common blocks are the ***initial*** block and the ***always*** block. A characteristic of these blocks is that they allow modeling behaviors through statements such as conditionals and loops.





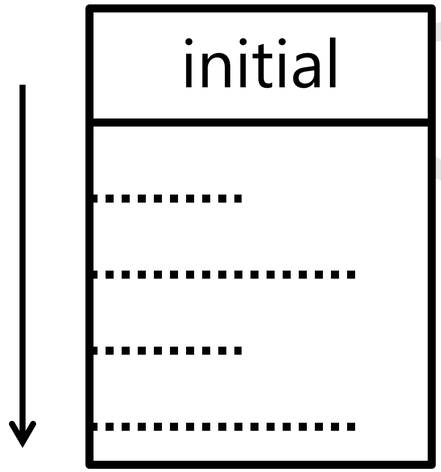



# Procedural assignments

- Procedural assignments are assignments made inside procedures.
- Procedural assignments can only be made to **variables** (NO nets).

# Initial

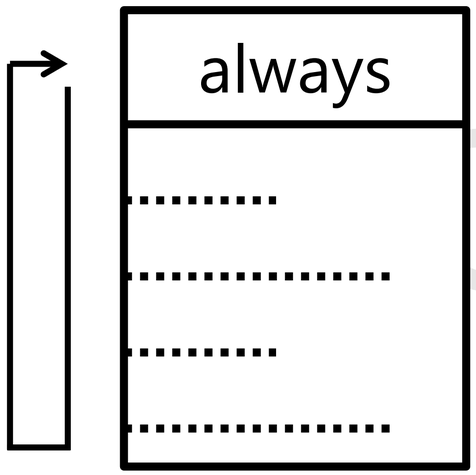
The initial block in Verilog is used to describe behavior that should **execute only once, at the beginning of the simulation**. All statements inside an initial block **start at time zero and run sequentially, like code in a software** program executes. It is commonly **used for testbenches, variable initialization, and setting up simulation conditions**, but it does not represent hardware that repeats or reacts continuously.




```
initial
begin
    a=0;
    #10;
    a=5;
    #10;
    a=3;
    #10;
end
```

# Always

Always construct is used to model behavior that should be executed **repeatedly** in response to changes in signals. Its execution is controlled by a **sensitivity list** (e.g., always @(posedge clk)), which **determines when the block should be triggered**. It is typically used to describe sequential logic (like flip-flops) and combinational logic, depending on how the **sensitivity list** is defined.

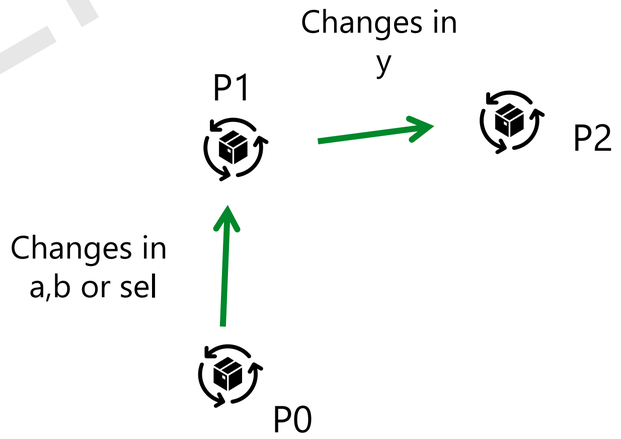



```
always @(*) //p1
begin
    if(sel)
        y = a;
    else
        y = b;
end
```

**Sensitive list**
Examples:
(a,b,sel)
(a or b)
(*) -- [all inputs]
(Posedge clk)
(Posedge clk or negedge rst)

# Interaction between assignments

```
module tb;
    reg a,b,sel;
    reg y;
    wire out;

    initial //p0
    begin
    a = 0;
    b = 0;
    sel = 0;
    #10;
    a = 0;
    b = 1;
    sel = 1;
    end

    always @(*) //p1
    begin
        if(sel)
            y = a;
        else
            y = b;
    end

    assign out = ~y; //p2

endmodule
```



# Procedural statements

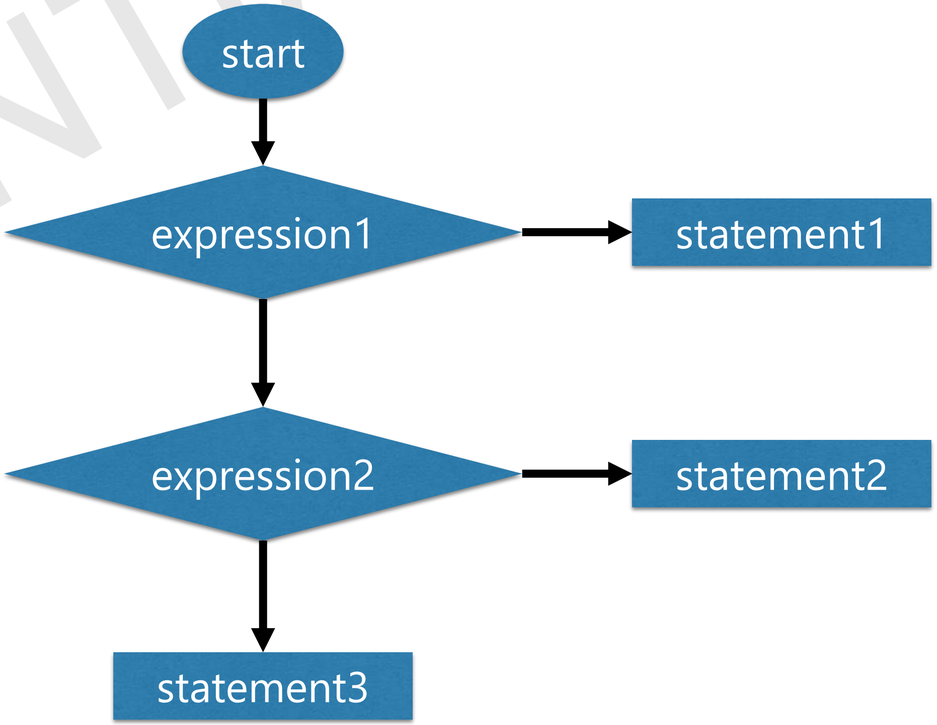
Procedural blocks can use:

- Conditional statements (if, else)
- Case statements (case, casez, casex)
- Loop statements (for, while repeat, forever)

# Procedural statements: Conditional

Conditional statements use if-else keywords to check conditions.
Syntax:

*if (expression1)*
*statement 1*
*[{else if (expression2)*
*statement2}]*
*[else*
*statement3]*






**Note:** Both **else if** and **else** statements are optional. However, if all possibilities are not specifically covered, **the synthesis will generate extra latches**, which are **undesirable** in most cases.

# Procedural statements: Conditional

Example

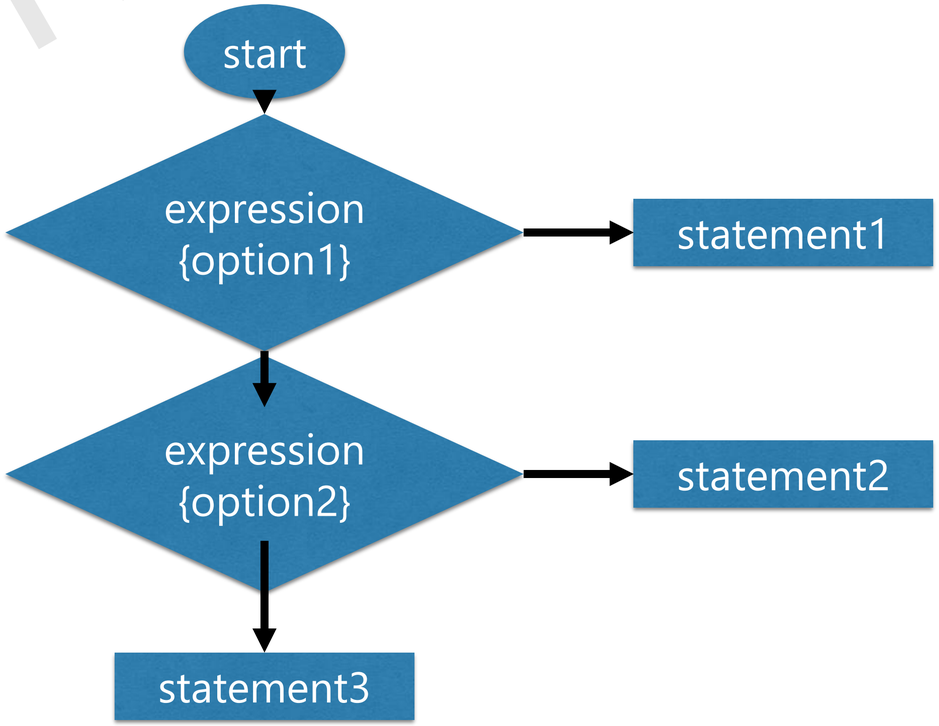
```
always @(*)
begin
    if(a == 2'b00) begin
        y0 = 1'b1;
        y1 = 1'b0;
        y2 = 1'b0;
        y3 = 1'b0;
    end else if(a == 2'b01) begin
        y0 = 1'b0;
        y1 = 1'b1;
        y2 = 1'b0;
        y3 = 1'b0;
    end else if(a == 2'b10) begin
        y0 = 1'b0;
        y1 = 1'b0;
        y2 = 1'b1;
        y3 = 1'b0;
    end else if(a == 2'b11) begin
        y0 = 1'b0;
        y1 = 1'b0;
        y2 = 1'b0;
        y3 = 1'b1;
    end
end
```

# Procedural statements: Case

The case statement generates a multi-branch decision based on the comparison of the expression with a list of options. The statements in the default block will be executed when none of the expressions are true.

*Syntax*

*case ( expression )*
*expression{option1} : statement1*
*{expression{option2} : statement2}*
*[ default [:] statement3]*
*endcase*




Without a default, **if no comparisons are true, synthesis will often generate unwanted latches**. Good design practices encourage the habit of including a default statement, whether it is strictly necessary or not.

# Procedural statements: case

Example

```
always @(*)
begin
case(a)
    2'b00:begin
        y = 4'b0001;
    end
    2'b01:begin
        y = 4'b0010;
    end
    2'b10:begin
        y = 4'b0100;
    end
    4'b11:begin
        y = 4'b1000;
    end
    endcase
end
```

# Procedural statements: case

Example

```
always @(*)
begin
case(a)
    2'b00:begin
        y = 4'b0001;
    end
    2'b01:begin
        y = 4'b0010;
    end
    2'b10:begin
        y = 4'b0100;
    end
    4'b11:begin
        y = 4'b1000;
    end
    endcase
end
```

Looks like a truth table!!!!

| a1 | a0 | y3 | y2 | y1 | y0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

# Procedural statements: case

How can we describe this logic?

| A3 | A2 | A1 | A0 | Y3 | Y2 | Y1 | Y0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | **1** |
| 0 | 0 | 1 | x | 0 | 0 | **1** | 0 |
| 0 | 1 | x | x | 0 | **1** | 0 | 0 |
| 1 | x | x | x | **1** | 0 | 0 | 0 |

# Procedural statements: casez

How can we describe this logic? casez!!!

```
always @(*)
begin
casez(a)
    4'b0000:begin
        y = 4'b0000;
    end
    4'b0001:begin
        y = 4'b0001;
    end
    4'b001?:begin
        y = 4'b0010;
    end
    4'b01??:begin
        y = 4'b0100;
    end
    4'b1???:begin
        y = 4'b1000;
    end
endcase
end
```

| A3 | A2 | A1 | A0 | Y3 | Y2 | Y1 | Y0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | **1** |
| 0 | 0 | 1 | x | 0 | 0 | **1** | 0 |
| 0 | 1 | x | x | 0 | **1** | 0 | 0 |
| 1 | x | x | x | **1** | 0 | 0 | 0 |

Can use don’t care values (? or Z)!!

# Procedural statements: casex

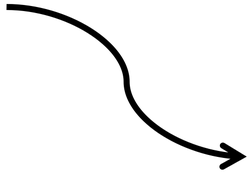
casex is similar to casez, the only difference is that **can handle x** states.

# Procedural statements: Loops

In Verilog has for, while repeat and forever loops. This are not recommendable for synthesis.

While loop syntax

```
while (expression)
    statement
```



The ***while*** loop executes its following statement while the expression is known and non-zero.

For loop syntax

```
for (initializer exp; condition exp; update exp)
    statement
```

The for loop performs the initial variable assignment and then while the test expression is known and non-zero executes the following statement.

# Procedural statements: Loops

In Verilog has for, while repeat and forever loops. This are not recommendable for synthesis.

Repeat loop syntax

```
repeat (expression)
    statement
```

A repeat loop executes its following statement the number of times specified by its parenthesized expression.

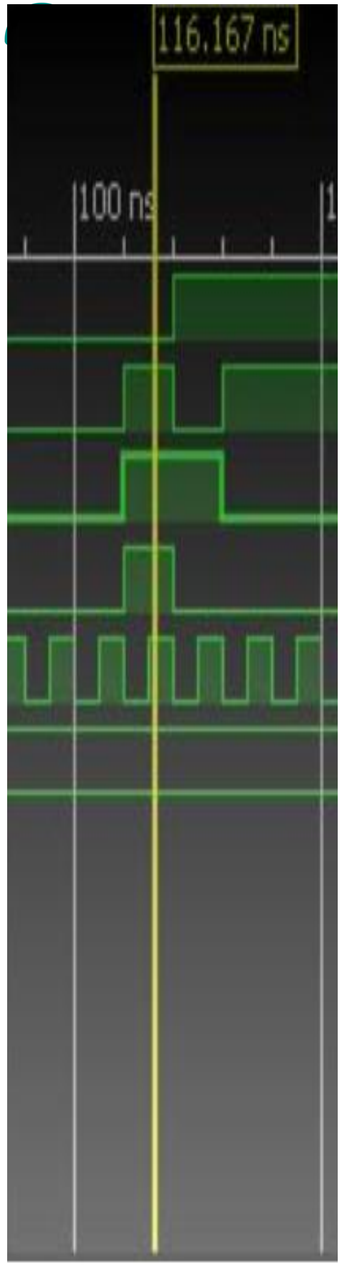
For loop syntax

```
forever
    statement
```

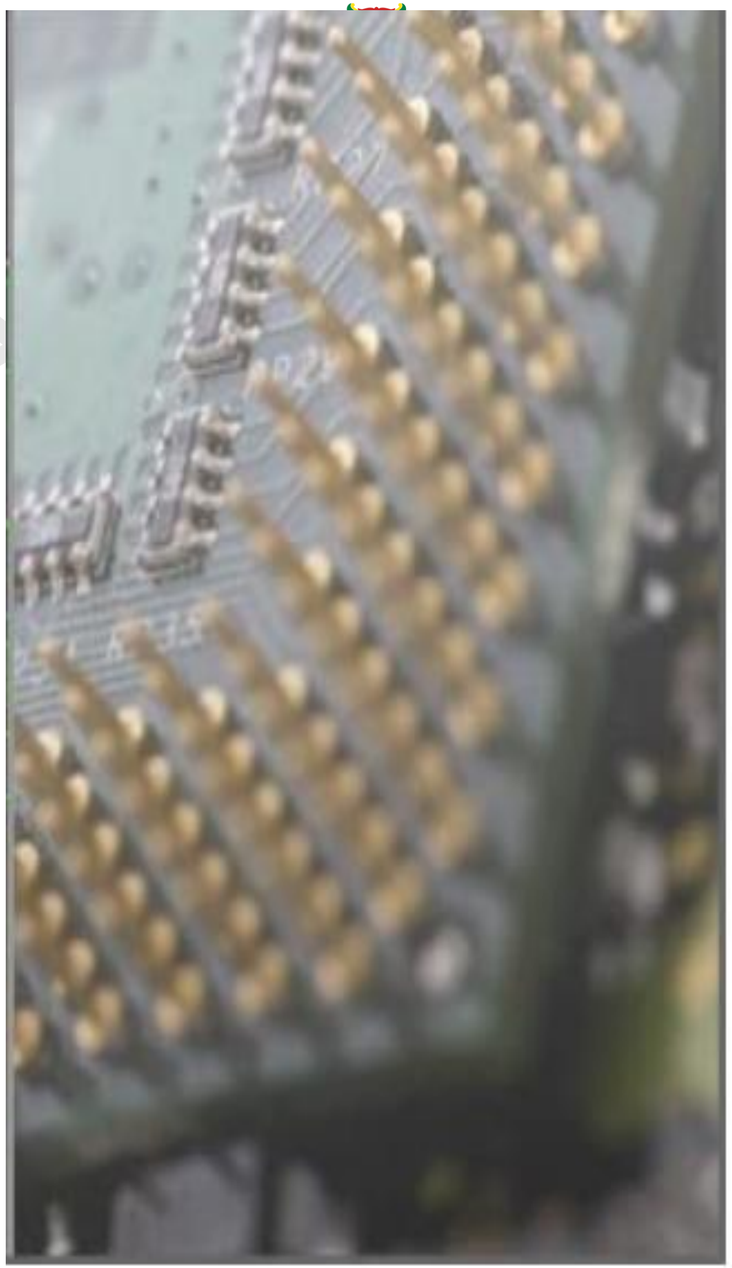
Forever loop repeat execution of a statement until simulation ends.

# LAB-1A: BCD to 7-segment display decoder

Design a BCD to 7-seg display decoder **using procedural blocks**.




# Generate statement

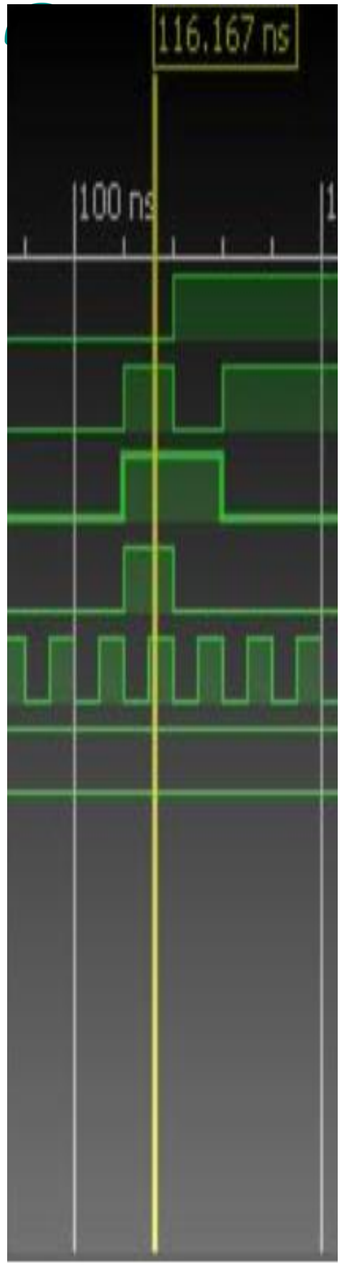


# Generate statement

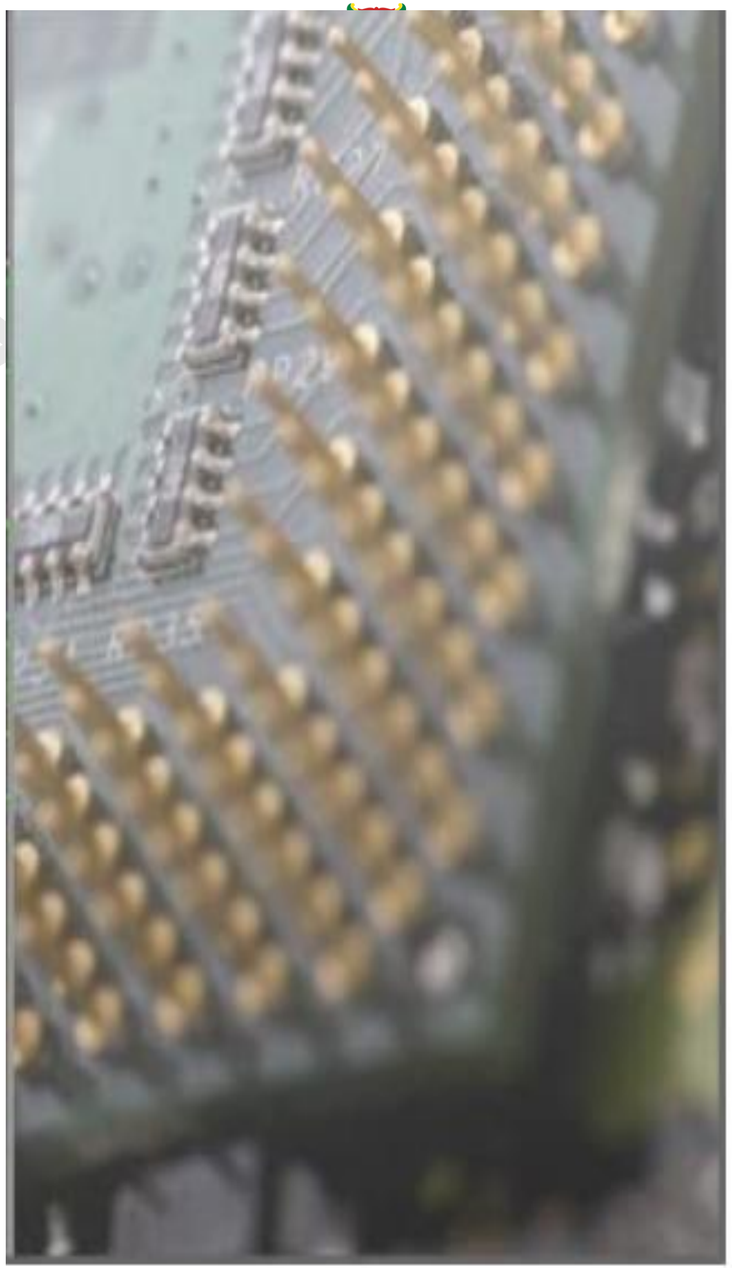
A **generate-for** loop allows one or more generate items to be instantiated multiple times. The loop index variable must be a **genvar**.
Inside of generate blocks we can use, variables, instances or procedural blocks (no function or tasks).

# Generate statement

```
module cla #(
    parameter SIZE = 4
) (
  input   [SIZE-1:0] a,
  input   [SIZE-1:0] b,
  input              ci,
  output  [SIZE-1:0] s,
  output             co
  );

  wire [SIZE-1:0] g;
  wire [SIZE-1:0] p;
  wire [SIZE:0] c;

  assign c[0] = ci;
  assign co   = c[n];

  genvar i;  /* i - generate index variable */

  generate
    for (i = 0; i < SIZE; i = i + 1) begin : addbit
      assign s[i] = a[i] ^ b[i] ^ c[i];
      assign g[i] = a[i] & b[i];
      assign p[i] = a[i] | b[i];
      assign c[i+1] = g[i] | (p[i] & c[i]);
    end
  endgenerate

endmodule
```

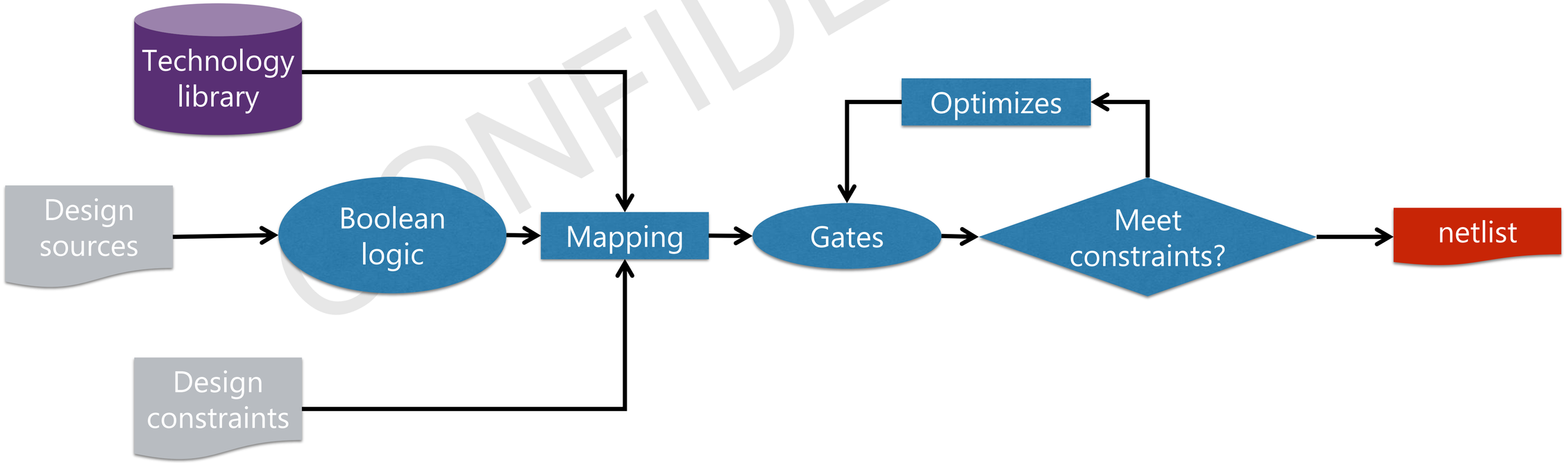



# Modeling Synthesizable logic



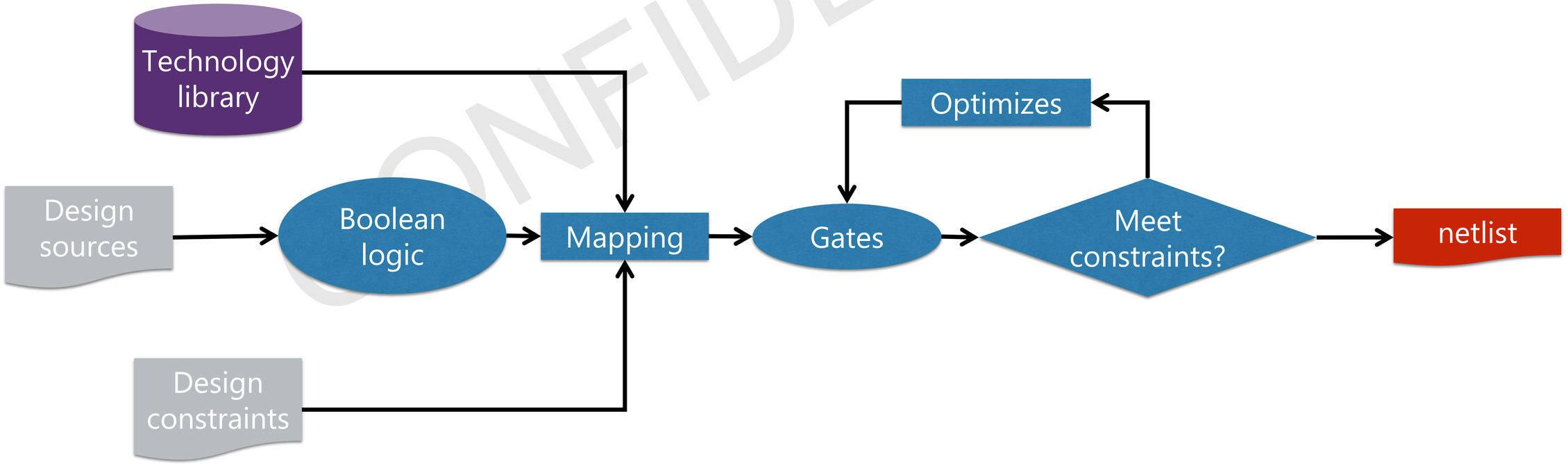
# What is logic synthesis?

Synthesis transform RTL into an optimized netlist of predefined cells.
We provide **design source**, **design constraints** and **technology library** to the tool, and it infers logic from the technology library and optimizes the circuit to meet constraints.



# Code matters !!

Synthesis tools don't read your mind. They transforms your RTL logic into a Boolean logic quite literally and then make optimizations. A good code can help to the process of optimization to obtain good results. Providing explicit descriptions of behavior can lead to more reliable hardware. **Remember, while you write RTL, what you are designing is hardware.**



# Code for synthesis

There are several guidelines for describing synthesizable RTL. The following are some recommendations for correctly describing hardware for both combinational and sequential logic.

# Combinational logic

Combinational logic is modeled as a process that is evaluated continuously whenever an input changes.

```
wire w = expression;

assign w = expression;

always * begin
    rega = expression;
end
```

# Combinational logic: sensitive list

"When using procedural blocks to describe combinational logic, it is important to include all inputs in the sensitivity list. The easiest way to achieve this is by using @*.

```
always @(a,b,sel) begin
    y = a;
    if (sel)
        y = b;
end
```

```
always @* begin
    y = a;
    if (sel)
        y = b;
end
```

# Combinational logic: Incomplete assignment

If an execution path in a procedure block is incomplete and one output is not updated, then that output value must be retained. In this case the synthesis tolls will infer a latch to model the behavior.

```
//latch
always @* begin
    if (sel)
        y = b;
end
```

```
//latch
always @* begin
    case (sel)
        1:y = b;
    endcase
end
```

# Combinational logic: Incomplete assignment

If an execution path in a procedure block is incomplete and one output is not updated, then that output value must be retained. In this case the synthesis tolls will infer a latch to model the behavior.

```
//latch
always @* begin
    if (sel)
        y = b;
end
```

```
//latch
always @* begin
    case (sel)
        1:y = b;
    endcase
end
```

Avoid latches is easy!!! Just complete the path or provide a default value.

```
//default value
always @* begin
    y = 1'b0;
    if (sel)
        y = b;
end
```

```
//complete path
always @* begin
    if (sel)
        y = b;
    else
        y = 1'b0;
end
```

```
//complete path
always @* begin
    case (sel)
        0:y = 1'b0;
        1:y = b;
    endcase
end
```

```
//default value
always @* begin
    y = 1'b0;
    case (sel)
        1:y = b;
    endcase
end
```