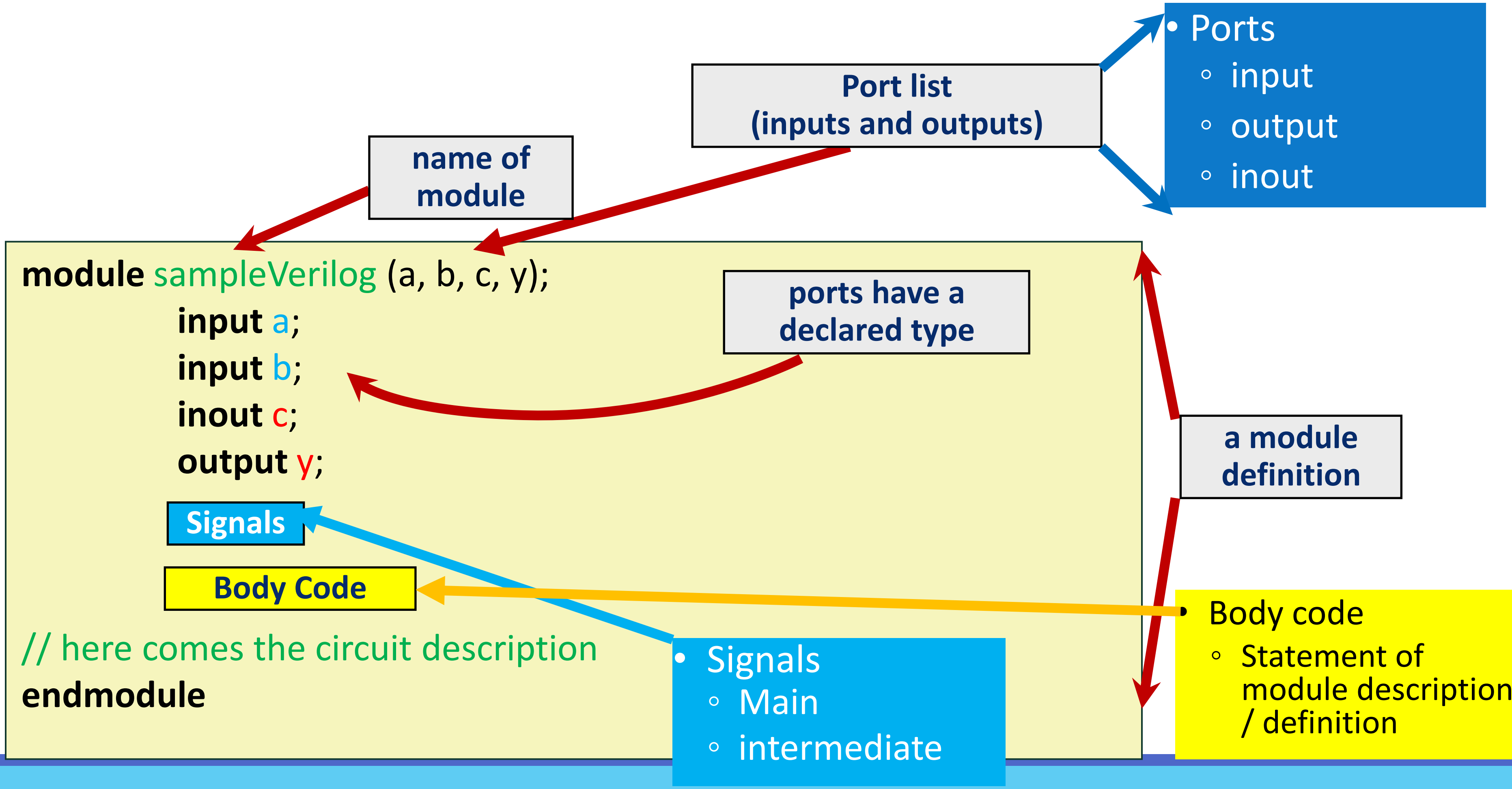
# Digital System Design

**Hajar Falahati**

hfalahati@ipm.ir
hfalahati@ce.sharif.edu

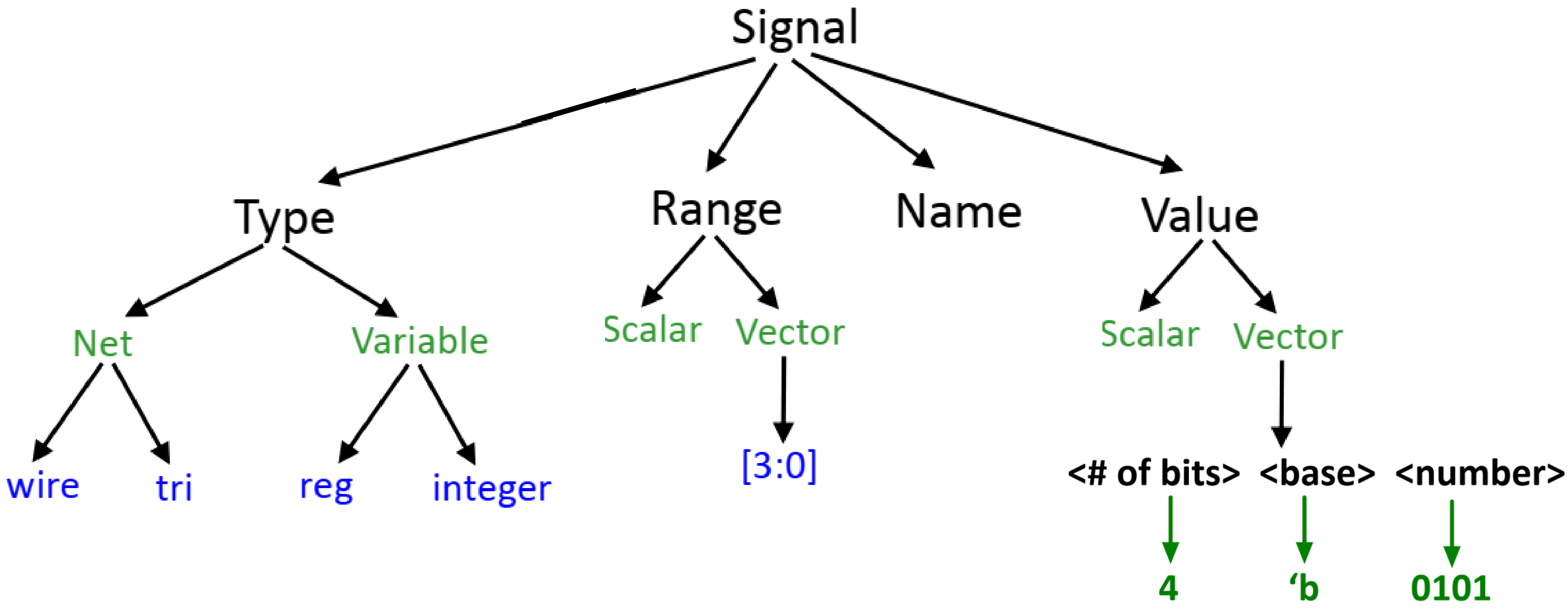
# Review: Module May Includes…

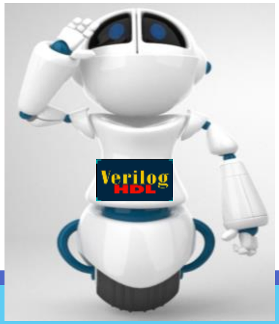


```
module sampleVerilog (a, b, c, y);
        input a;
        input b;
        inout c;
        output y;
        Signals
        Body Code
// here comes the circuit description
endmodule
```

- Ports
  - input
  - output
  - inout
- Signals
  - Main
  - intermediate
- Body code
  - Statement of module description / definition

# Review: Signal Value





# Review: Verilog Statement

**Verilog Statements**

**Procedural statement**

**Evaluated sequentially**
**Order IS important**

```
always @ (x, y)
  begin
    s = x ^ y;
    c = x & y;
  end
```

**Concurrent statement**

**Evaluated in parallel**
**Order is NOT important**

```
assign  s = x ^ y;
assign c = x & y;
assign  out = x | y;
```

## Outline

- Behavioral-level programming
  - Initial blocks
  - Always Blocks
- How to implement a combinational logic at behavioral-level?
- How to implement a sequential logic at behavioral-le
- Writing a testbench

## Verilog:
## A language for All Abstraction Levels!

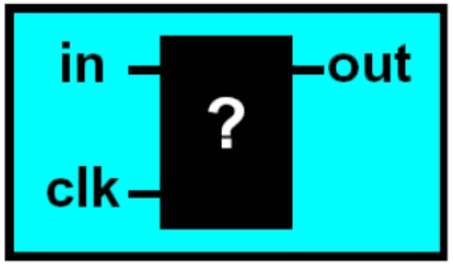
## Behavioral

- Describe the behavior of a design without implying any specific internal architecture
- Defile the behavior of the design as seen as its ports
  - Using high-level constructs
    - @, case, if, repeat, wait, while
  - Using any behavioral construct of the HDL in your testbench
  - Synthesis tools accept only a limited subset of these behavioral constructs

# Behavioral Code

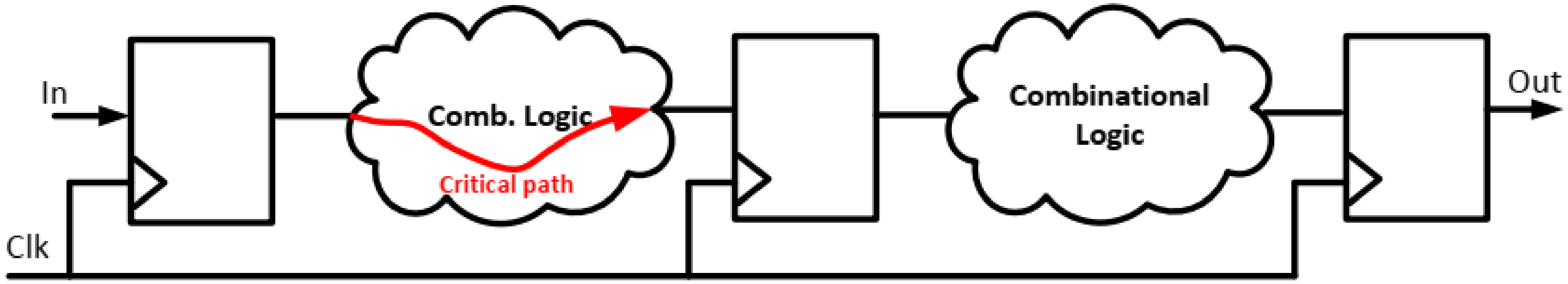
```
module sampleBehavioral (out, in, clk);
        input in, clk;
        output out;
        reg out;

        always @(in)
             @(posedge clk)
                  out  <=repeat (2) @(posedge clk) in;
endmodule
```

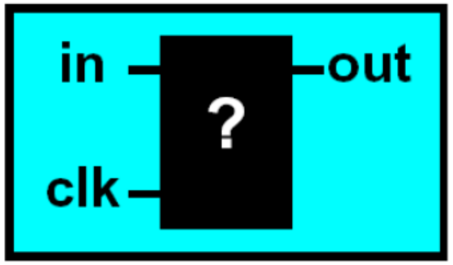


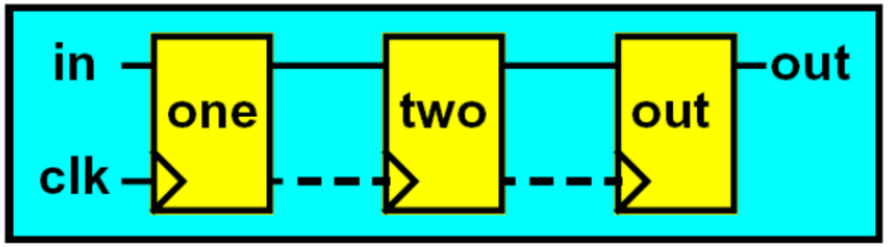
# RTL

- Functional level
- Describe a design architecture in sufficient detail that a synthesis tool can construct the circuit
- RTL is the closet one to the actual hardware implementation
- RTL code includes a subset of all Verilog syntax
  - Not all Verilog syntax are synthesizable



# RTL Code



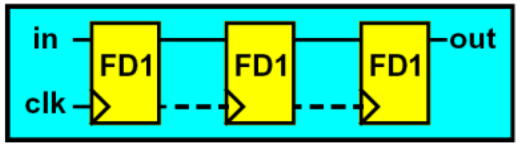


```
module sampleBehavioral (out, in, clk);
  input in, clk;
  output out;
  reg out;
always @(in)
  @(posedge clk)
  out<=repeat (2) @(posedge clk) in;
endmodule
```

```
module sampleRTL (out, in, clk);
        input in, clk;
        output out;
        reg out;
        reg one, two;
        always @(posedge clk)
          begin
                out <= two;
                two <= one;
                one <= in;
          end
endmodule
```

# Structural Recall

- Structural design description
- Appropriate for small library components
  - Using built-in Verilog primitives, such as the and gate
  - Describe your own User defined primitives (UDPs)



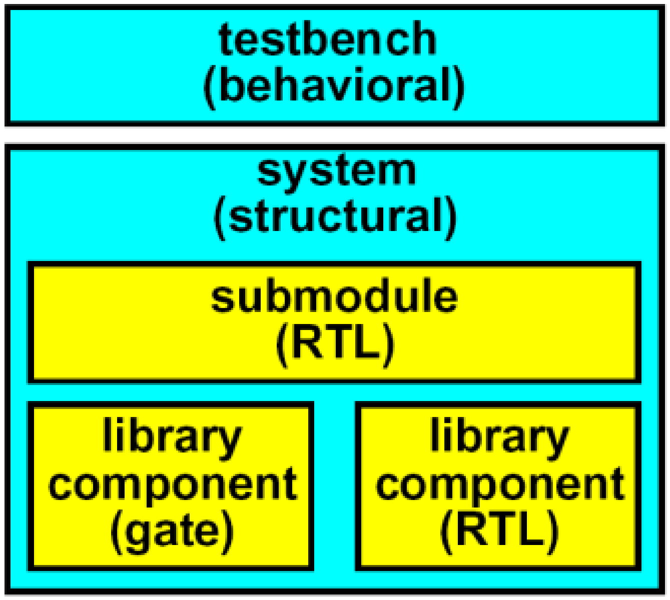
```
module sampleStructural (out, in, clk);
        input in, clk;
        output out;

FD1 one_reg (.Q(one),.D(in),.CP(clk);
FD1 one_reg (.Q(two),.D(one),.CP(clk);
FD1 one_reg (.Q(out),.D(two),.CP(clk);

endmodule
```

# Verilog: A language for All Abstraction Levels!

- Designers usually mix levels of abstraction within a simulation
- RTL and gate-level library components
- RTL functional submodule descriptions
- Structural system netlist
- Behavioral system testbench

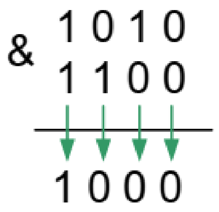


## Operators

# Operators 1, 2

**Bitwise:**

| Operation | Result |
|---|---|
| 1010 **&** 1100 | 1000 |
| 1010 **\|** 1100 | 1110 |
| **~**1010 | 0101 |
| 1101 **^** 0100 | 1001 |



**Logical:**

X || 1 = 1
X && 0 = 0

| Operation | Result |
|---|---|
| 1010 **&&** 1100 | 1 |
| 2'b11 **\|\|** 2'b00 | 1 |
| **!**0010 | 0 |
| 2'b1X **&&** 2'b11 | X |

Non-zero operand=logical "1"

Any operand X/Z, result is X

## Operators 3, 4

**Reduction:**

| Operation | Result |
|---|---|
| & 1100 | 0 |
| & 111 | 1 |
| ^ 0100 | 1 |

**Relational:**

A=2'b10

| Operation | Result |
|---|---|
| B=(A == 2'b10) | B=1 |
| B=(A == 2'b11) | B=0 |
| B=(A === 2'b1x) | B=0 |
| B=(A <= 2'b11) | B=1 |

== Used only with 0 and 1

=== Used with x and z

## Operators 5, 6

**Logical Shift:**

A=6'b001100

| Operation | Result |
|---|---|
| C = A >> 1 | C = 000110 |
| D = A << 2 | D = 110000 |
| F = A >> 3 | F = 000001 |

**Concatenation:**

A=2'b11
B=3'b010

| Operation | Result |
|---|---|
| {A, B} | 5'b11010 |
| {3{A}} | 6'b111111 |
| {B, B} | 6'b010010 |
| {{3{A}}, {2{B}}} | 12'b111111010010 |

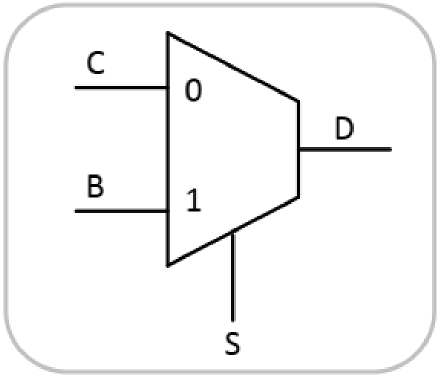
Be generous in {} $\nearrow$

# Operators 7

**Conditional: (? , : )**

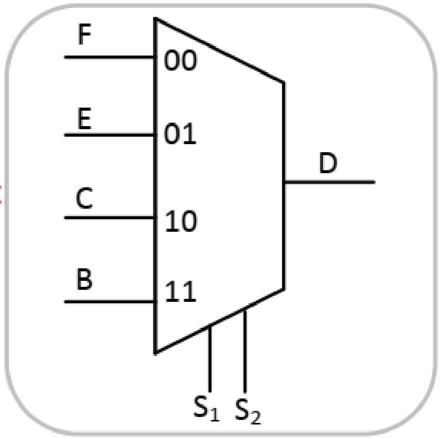
- D = S ? B:C;

$$D = \begin{cases} B & \text{if } S=1; \\ C & \text{if } S=0; \end{cases}$$



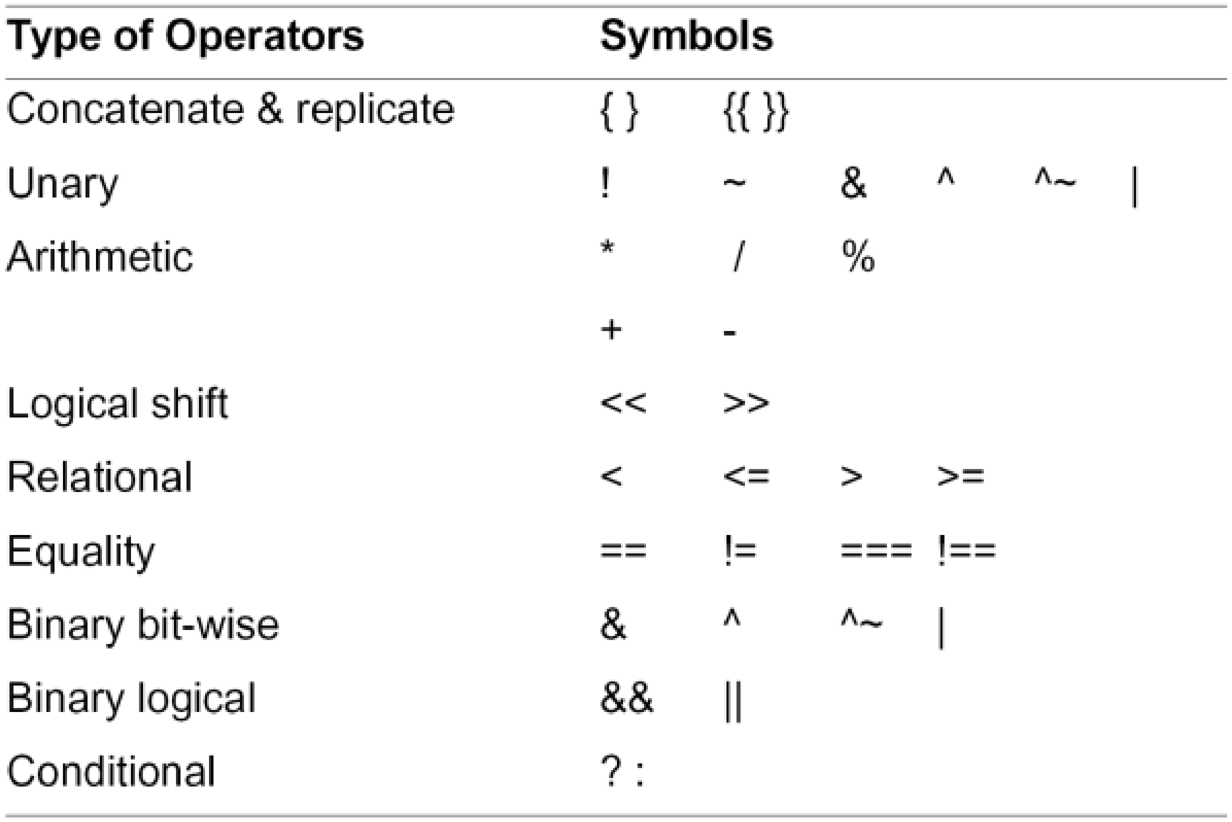
- D = ({$S_1$,$S_2$}==2'b00)? F:
  ({$S_1$,$S_2$}==2'b01)? E:
  ({$S_1$,$S_2$}==2'b10)? C:B;
- D = ({$S_1$,$S_2$}==2'b00)? F:
  ({$S_1$,$S_2$}==2'b01)? E:
  ({$S_1$,$S_2$}==2'b10)? C:
  ({$S_1$,$S_2$}==2'b11)? B:B;

Default



**4-input Multiplexer (MUX)**

# Operators: All in one

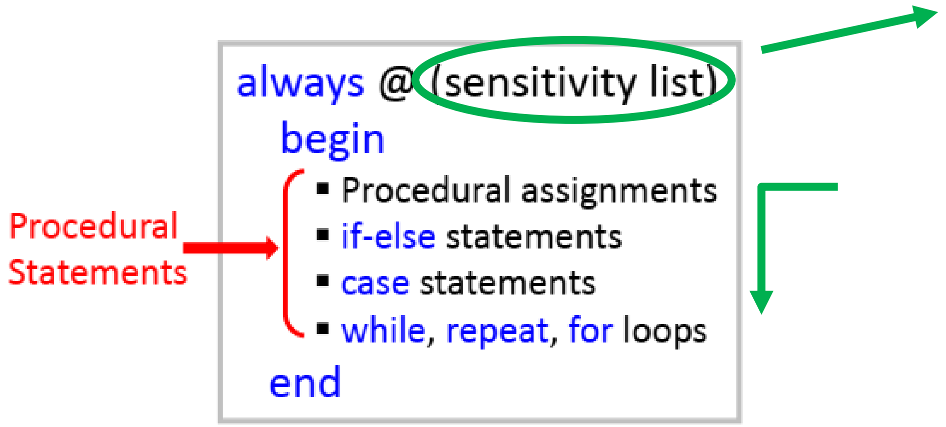


| Type of Operators | Symbols | | | | | |
|---|---|---|---|---|---|---|
| Concatenate & replicate | { } | {{ }} | | | | |
| Unary | ! | ~ | & | ^ | ^~ | \| |
| Arithmetic | * | / | % | | | |
| | + | - | | | | |
| Logical shift | << | >> | | | | |
| Relational | < | <= | > | >= | | |
| Equality | == | != | === | !== | | |
| Binary bit-wise | & | ^ | ^~ | \| | | |
| Binary logical | && | \|\| | | | | |
| Conditional | ? : | | | | | |

## Behavioral-Level

# Procedure Statements

- Evaluated in the order in which they appear in the code
  - Sequential
- Should be inside an "always" block
- An "always" block contains one or more procedural statements
- No nesting

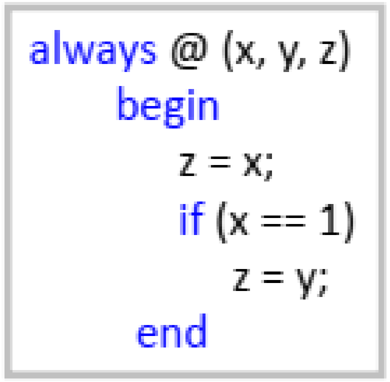


```
always @ (sensitivity list)
   begin
      ▪ Procedural assignments
      ▪ if-else statements
      ▪ case statements
      ▪ while, repeat, for loops
   end
```

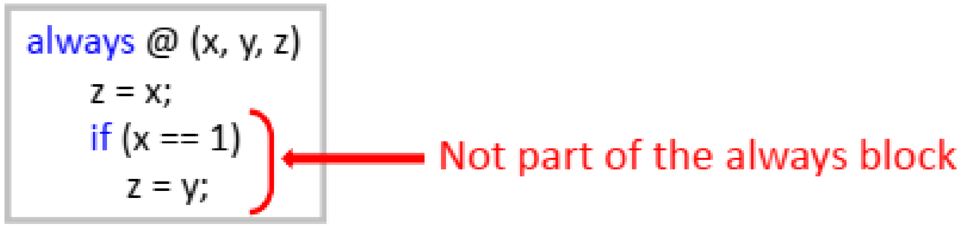
List of all signals that trigger the evaluation inside the always block

# Always Block: Structure

- Requires **begin-end** only of there are multiple statements in the block



```
always @ (x, y, z)
    begin
        z = x;
        if (x == 1)
            z = y;
    end
```



```
always @ (x, y, z)
    z = x;
    if (x == 1)
      z = y;
```

## Always Block Value

- Any signal assigned inside an always block **have to** be a variable of type
  - reg
  - integer

- **i.e.,** value remains unchanged until another procedural assignment updates it

**< value> = <expression >**

# Always Block Value (cont’d)

**< value> = <expression >**

- *value*
  - reg, integer, real, time
  - A bit-select of the above (e.g., addr[0])
  - A part-select of the above (e.g., addr[31:16])
  - A concatenation of any of the above

- *expression*
  - Any type (register, net) or function

- What happens if the widths do not match?
  - LHS wider than RHS
    - RHS is zero-extended
  - RHS wider than LHS
    - RHS is truncated (Least significant part is kept)

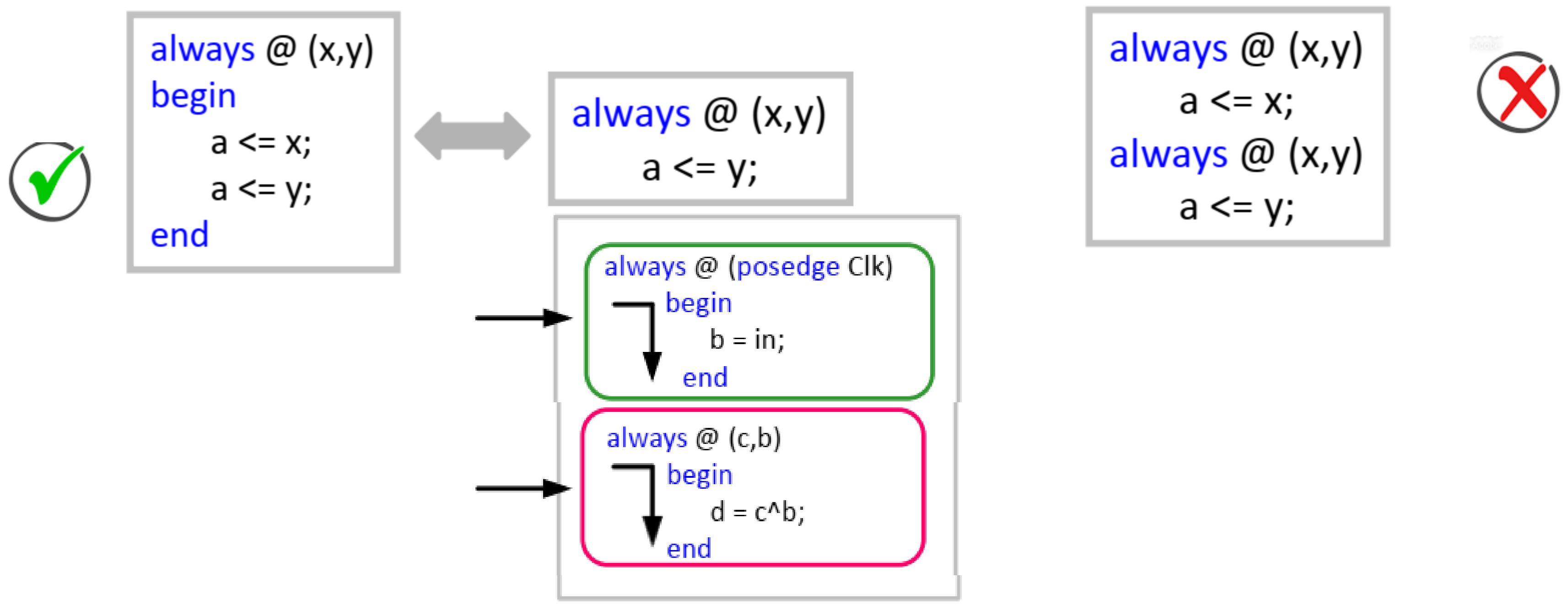
## Always Block Value (cont'd)

**< value> = <expression >**

- What happens if the widths do not match?
  - LHS wider than RHS
    - RHS is zero-extended
  - RHS wider than LHS
    - RHS is truncated (Least significant part is kept)

# Always Block: Rule

- A given variable should **never be assigned** a value in more than one always block
  - Always blocks are concurrent with respect to one another
  - Start from simulation time 0



```
always @ (x,y)
begin
    a <= x;
    a <= y;
end
```

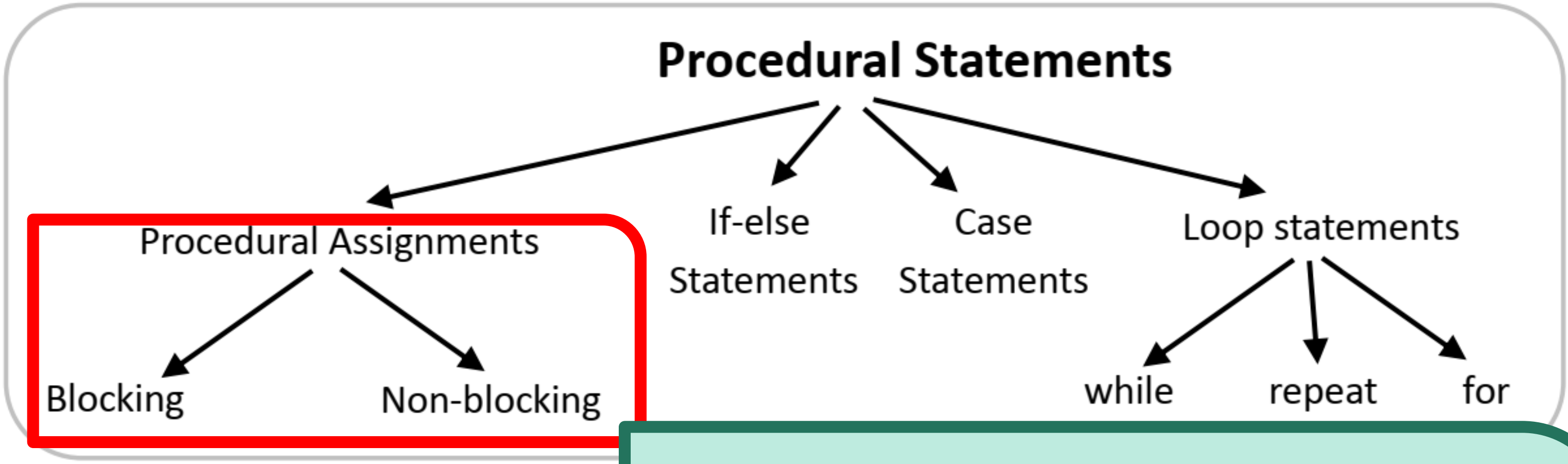
```
always @ (x,y)
    a <= y;
```

```
always @ (x,y)
    a <= x;
always @ (x,y)
    a <= y;
```

```
always @ (posedge Clk)
    begin
        b = in;
    end
```

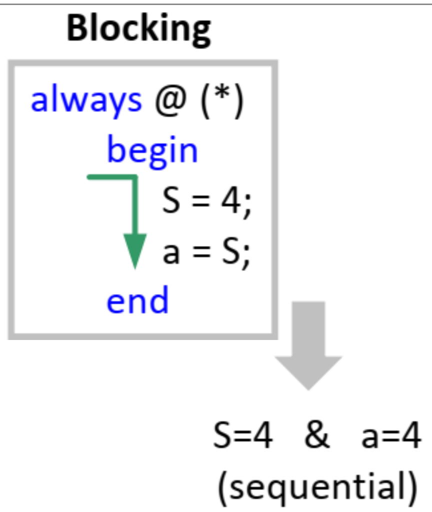
```
always @ (c,b)
    begin
        d = c^b;
    end
```

# Procedural Statement Types



## Procedural Assignment

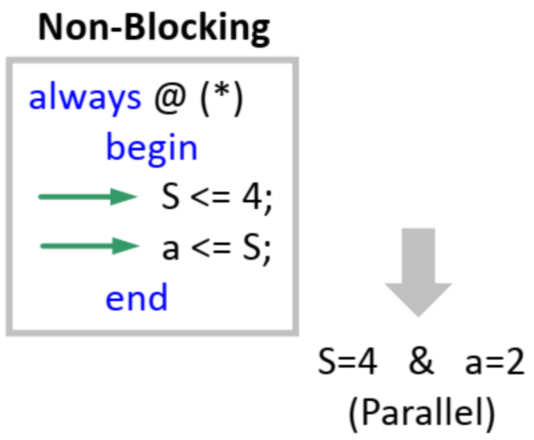
- Assume S=2



**Blocking**

```
always @ (*)
    begin
        S = 4;
        a = S;
    end
```

S=4 & a=4
(sequential)



**Non-Blocking**

```
always @ (*)
    begin
        S <= 4;
        a <= S;
    end
```

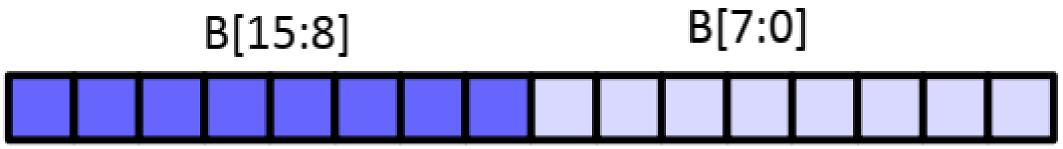
S=4 & a=2
(Parallel)

- Evaluated and assigned in a single step
- Sequential nature
- Assignment ordering is important

- Evaluated and assigned in two steps
  - All RHSs are evaluated in parallel
  - Assignment to LHSs are performed together
- They are evaluated all at once
- Assignment ordering is NOT important
- S <=4 and a<=S evaluated in parallel

# Procedural Assignment: Sample

- Swap bytes in words

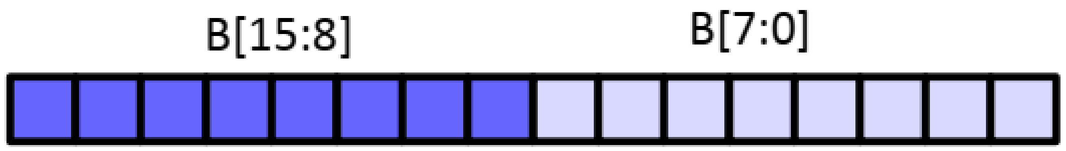


- Evaluated and assigned in a single step
- Sequential nature
- Assignment ordering is important

- Evaluated and assigned in two steps
  - All RHSs are evaluated in parallel
  - Assignment to LHSs are performed together
- They are evaluated all at once
- Assignment ordering is NOT important
- S <=4 and a<=S evaluated in parallel

# Procedural Assignment: Swap bytes in words

- Which one is correct?



**Blocking**

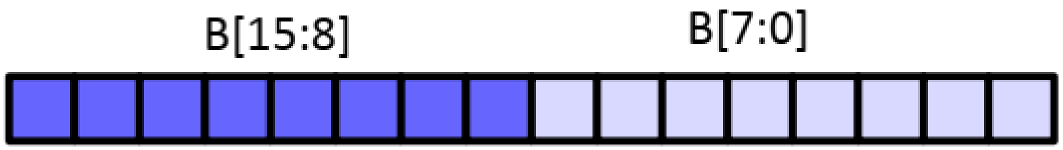
```
always @ (*)
      begin
            B[15:8] = B[7:0];
            B[7:0]   = B[15:8] ;
      end
```

**Non-Blocking**

```
always @ (*)
      begin
            B[15:8] <= B[7:0];
             B[7:0]   <= B[15:8] ;
      end
```

# Procedural Assignment: Swap bytes in words ?

- Which one is correct?

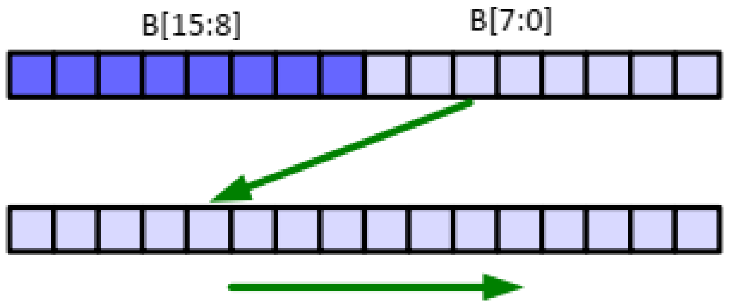


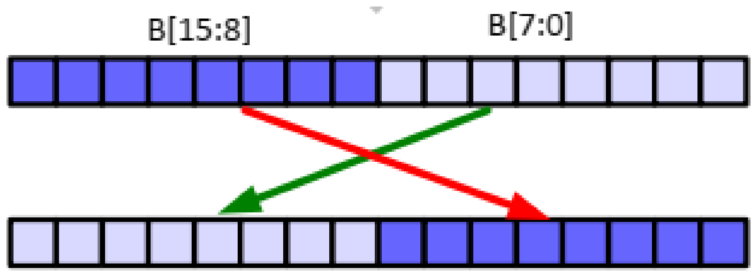
**Blocking**

```
always @ (*)
     begin
          B[15:8] = B[7:0];
          B[7:0]   = B[15:8] ;
     end
```

**Non-Blocking**

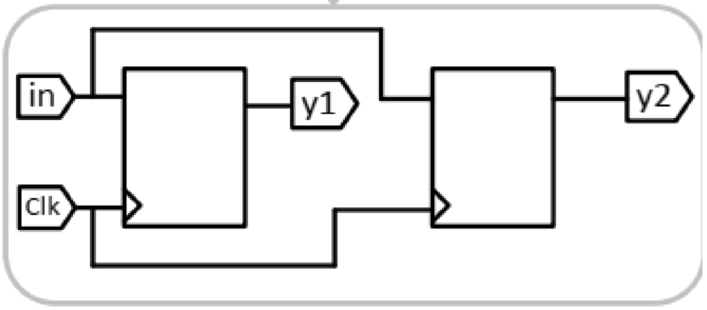
```
always @ (*)
     begin
          B[15:8] <= B[7:0];
           B[7:0]   <= B[15:8] ;
      end
```

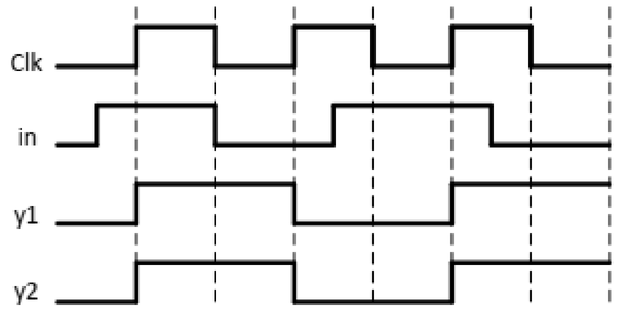




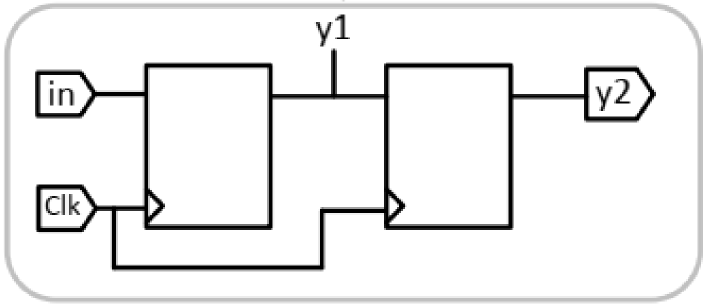
# Procedural Assignment: After Synthesis

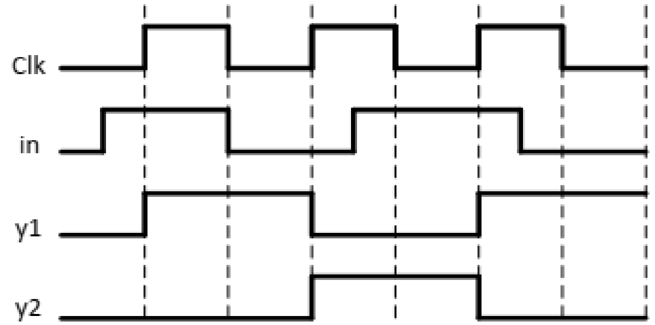
```
always @ (posedge Clk)
    begin
        y1 = in;
        y2 = y1 ;
    end
```



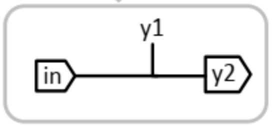


```
always @ (posedge Clk)
    begin
        y1 <= in;
        y2 <= y1 ;
    end
```

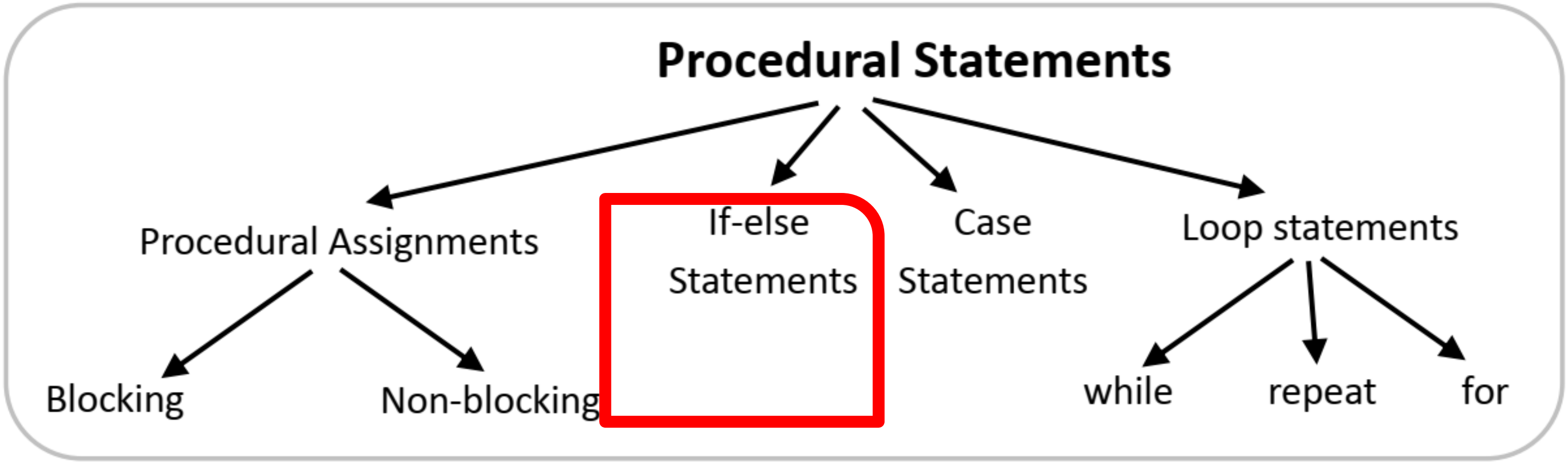


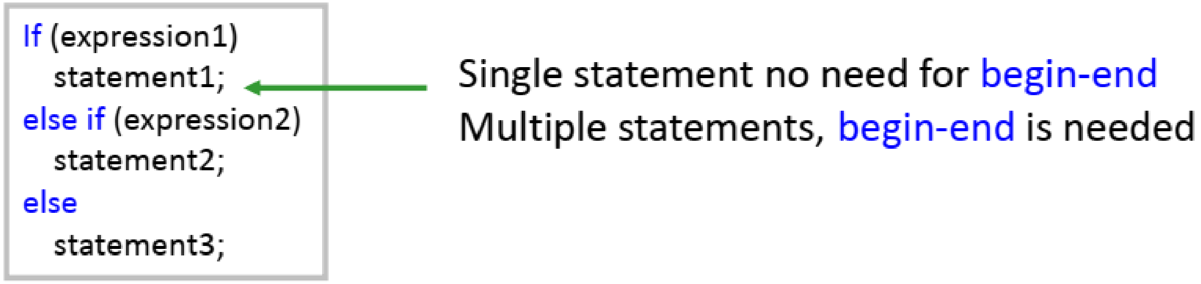


```
always @ (*)
    begin
        y1 = in;
        y2 = y1 ;
    end
```



# Procedural Statement Types

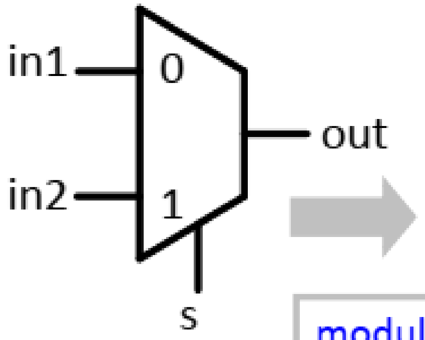




```
If (expression1)
   statement1;
else if (expression2)
   statement2;
else
   statement3;
```

Single statement no need for begin-end
Multiple statements, begin-end is needed

# Procedural Statement: If-else Sample

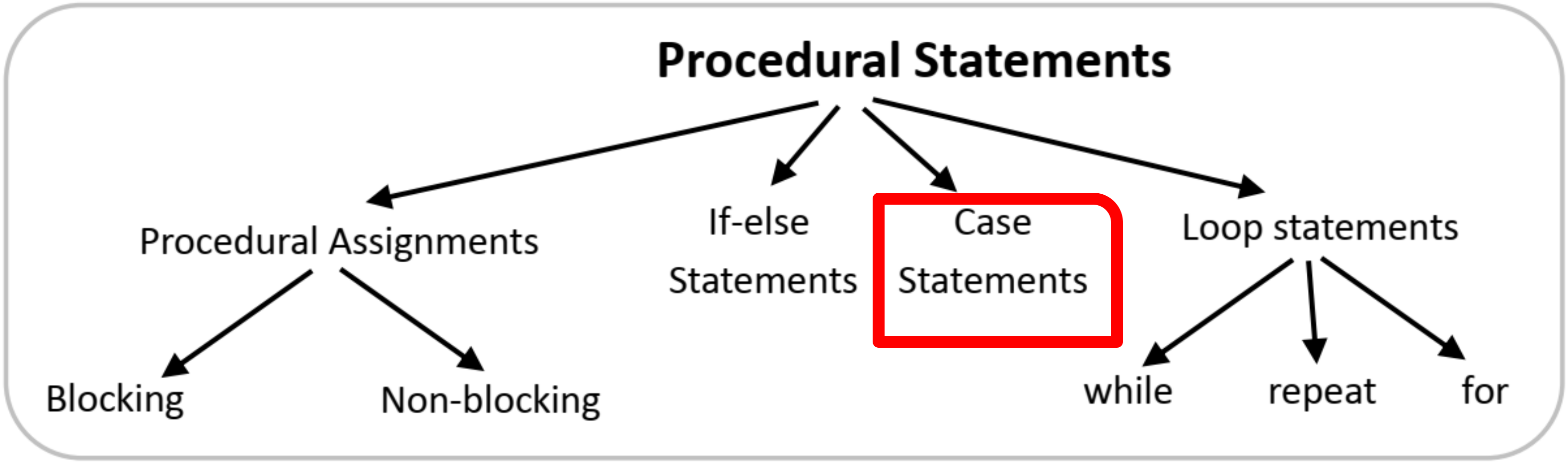


```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
       if (s==0)
           out = in1;
       else
           out = in2;
endmodule
```

```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;
   always @ (in1, in2, s)
       begin
           out = in1;
           if (s==1)
               out = in2;
       end
endmodule
```

# Procedural Statement Types

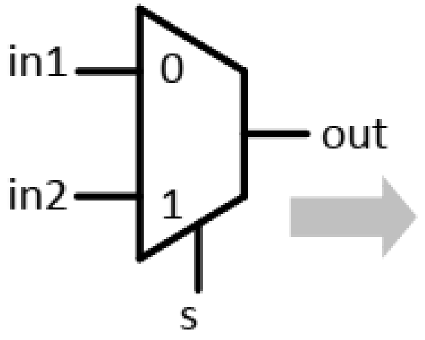


```
case (expression)
   alternative1: statement1;
   alternative2: begin
                     statement2;
                 end
   default: statementn;
endcase
```

Single statement no need for begin-end

Multiple statements, begin-end is needed

# Procedural Statement: Case Statement Sample



```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
      case (s)
         1'b0: out = in1;
         1'b1: out = in2;
      endcase
endmodule
```

```
module Mux21 (in1, in2, s, out)
   input in1, in2, s;
   output reg out;

   always @ (in1, in2, s)
      case (s)
         1'b0: out = in1;
         default: out = in2;
      endcase
endmodule
```

# Procedural Statement: Case Statement: don’t care

In case statements, each alternative is compared for an exact match

Synthesis tools are only concerned about matching of “0” and “1” while “Z” and “X” are not important

If “X” or “Z” are needed to be added, casex is used (casex is synthesizable)

In fact casex treats them as don’t care

| w3 | w2 | w1 | w0 | y1 | y0 | f |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | d | d | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 1 | 1 |
| 0 | 1 | X | X | 1 | 0 | 1 |
| 1 | X | X | X | 1 | 1 | 1 |

→

```
module Priority (W, Y, f)
   input [3:0] W;
   output reg [1:0] Y;
   output f;
   assign f = (W!=0)
   always @ (W)
     begin
       casex (W)
           'b1xxx:  Y = 3;
           'b01xx:  Y = 2;
           'b001x:  Y = 1;
           default: Y = 0;
       endcase
     end
endmodule
```

## Procedural Statement: Case Statement: don’t care

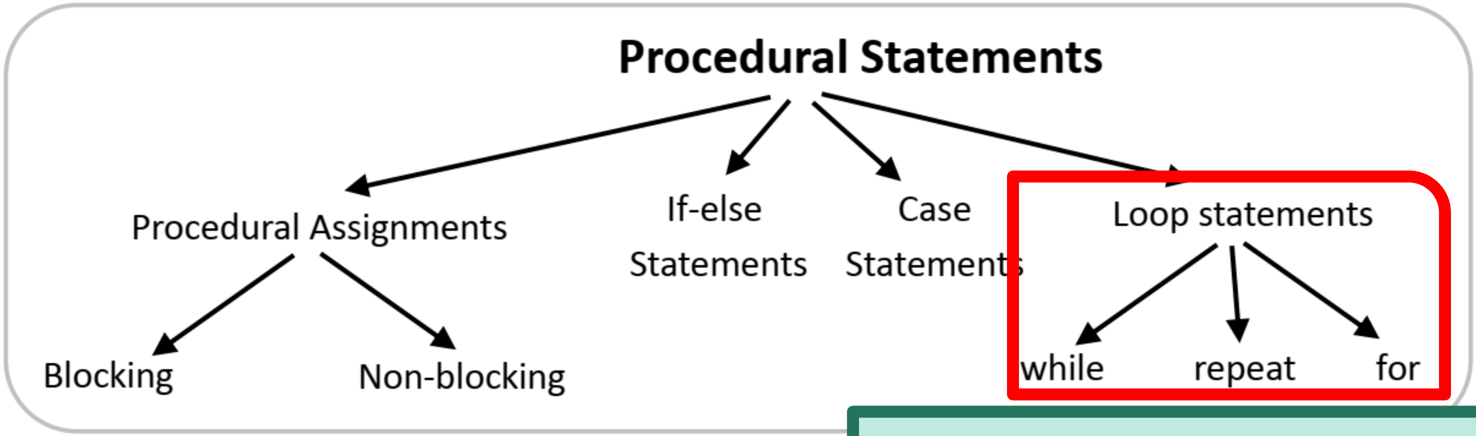
**casez** allows use of wildcard “?” character for don’t

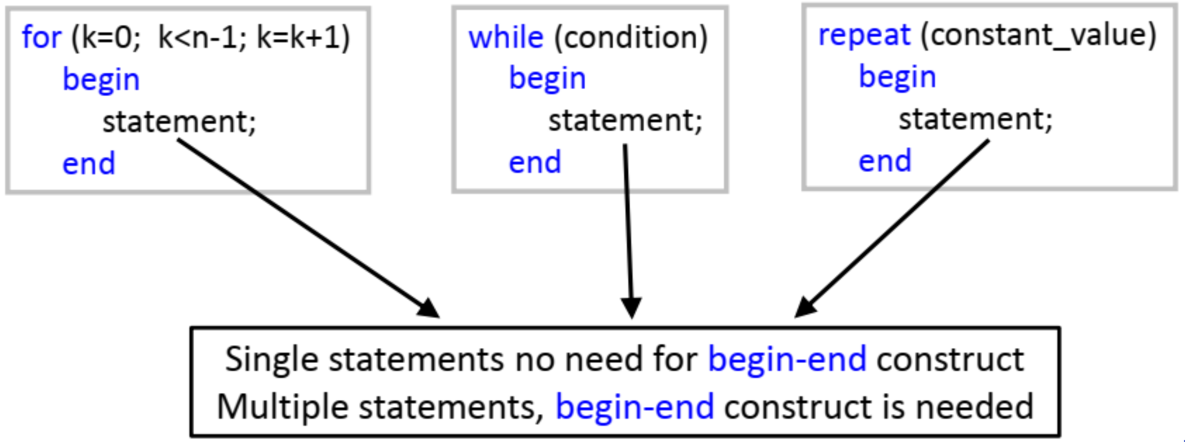
| w3 | w2 | w1 | w0 | y1 | y0 | f |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | d | d | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 1 | 1 |
| 0 | 1 | X | X | 1 | 0 | 1 |
| 1 | X | X | X | 1 | 1 | 1 |

→

```
module Priority (W, Y, f)
   input [3:0] W;
   output reg [1:0] Y;
   output f;
   assign f = (W!=0)
   always @ (W)
      begin
         casez (W)
              ’b1???:  Y = 3;
              ‘b01??:  Y = 2;
              ‘b001?:  Y = 1;
              default: Y = 0;
         endcase
      end
endmodule
```

# Procedural Statement Types





```
for (k=0;  k<n-1; k=k+1)
   begin
      statement;
   end
```

```
while (condition)
   begin
      statement;
   end
```

```
repeat (constant_value)
   begin
      statement;
   end
```

Single statements no need for begin-end construct
Multiple statements, begin-end construct is needed

# Sample 1:

How to implement a function like: F(a,b,c,d) = ab + cd?

```
module F (input a,b,c,d, output
reg out);

always @(a,b,c,d)

begin

        out = (a & b) | (c & d);

end

endmodule
```

```
module F (input a,b,c,d, output
reg out);

always @(*)

begin

        out = (a & b) | (c & d);

end

endmodule
```

# Write a testbench for “F” module

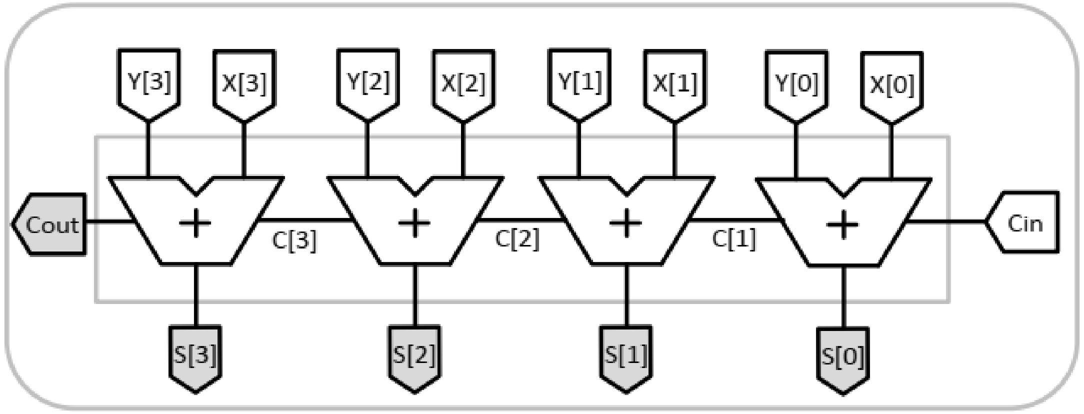
```
module testbench;
reg a, b, c, d;
wire out;
F inst (a, b, c, d, out);
initial
begin
{a,b,c,d} = 4’b0001;
#5
{a,b,c,d} = 4’ha;
endmodule
```

# Sample 2:

N-bit ripple carry adder

```
module RippleCarryAdderI (Cin, X, Y, S, Cout)
  parameter n = 4;
  input  Cin;
  input [n-1:0] X, Y;
  output  reg [n-1:0] S;
  output reg  Cout;
  reg [n:0] C;
  integer k;
  always  @(X, Y, Cin)
    begin
      C[0] = Cin;
      for (k=0;k<=n-1;k=k+1)
        begin
          S[k] = X[k] ^ Y[k] ^ C[k];
          C[k+1] = (X[k] & Y[k])
                 |(X[k] & C[k])|(Y[k] & C[k]);
        end
      Cout = C[n];
    end
endmodule
```

**Sequential Structure**



Breaking one statement in two lines is allowed!

## Thank You