System Specification Using Verilog HDL

Number Specification

- When specifying constants, whether they be single bit or multi-bit, you should use an explicit syntax to avoid confusion:

The general syntax is:

- {bit width}'{base}{value}
- 4'd14 // 4-bit value, specified in decimal
- 4'he // 4-bit value, specified in hex
- 4'b1110 // 4-bit value, specified in binary
- 4'b10xz // 4-bit value, with x and z, in binary

Delay Control

You can add control the timing of assignments in procedural blocks in several ways:

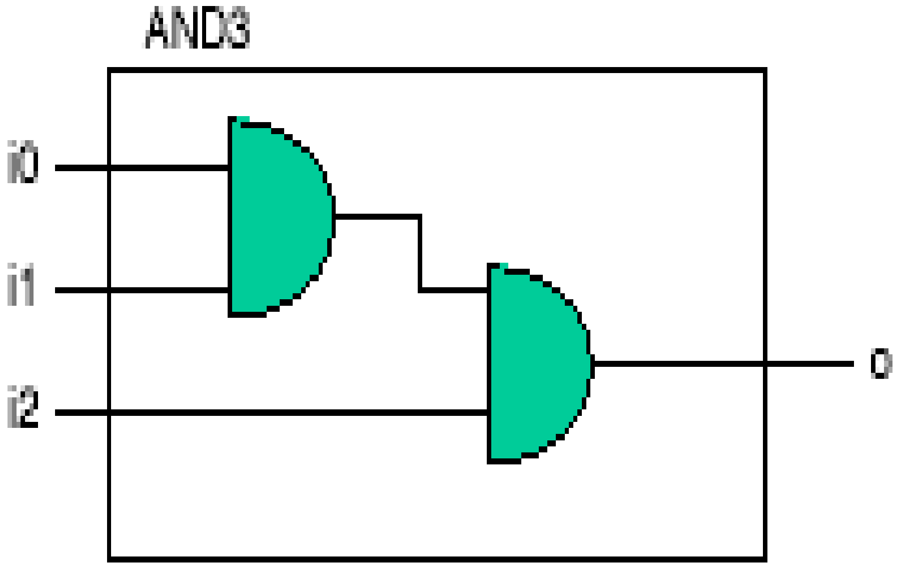
- Simple delays.
  - #10;
  - #10 a = b;
- Edge triggered timing control.
  - @(a or b);
  - @(a or b) c = d;
  - @(posedge clk);
  - @(negedge clk) a = b;

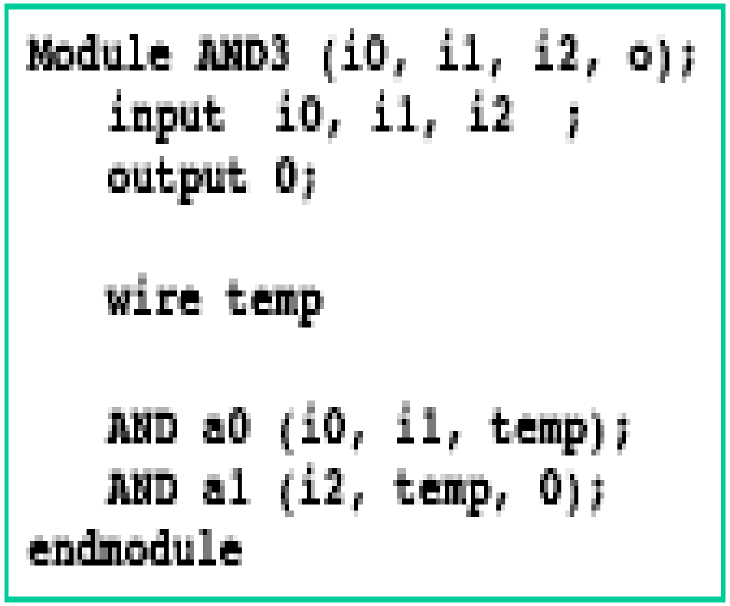
- Example:

```
`timescale 1ns/100ps
module HalfAdder (A, B, Sum, Carry);
  input A, B;
  output Sum, Carry;
  assign #3 Sum = A ^ B;
  assign #6 Carry = A & B;
endmodule
```

Hierarchical Modeling Structure

- Module instances
  - Verilog models consist of a hierarchy of module *instances*





```
Module AND3 (i0, i1, i2, o);
    input  i0, i1, i2  ;
    output 0;

    wire temp

    AND a0 (i0, i1, temp);
    AND a1 (i2, temp, 0);
endmodule
```

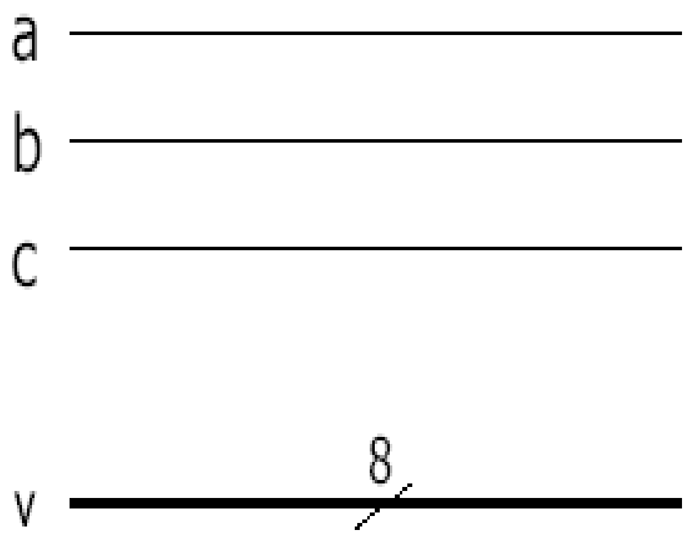
Data Type

- There are two main data types in Verilog. These data types may be single bit or multi-bit.
  - Wires
    - ✓ Wires are physical connections between devices and are "continuously assigned".
    - ✓ Nets do not "remember", or store, information -This behaves much like an electrical wire...
  - Registers
    - ✓ Regs are "procedurally assigned" values and "remember", or store, information until the next value assignment is made.
    - ✓ It is not hardware register

Data Type Declaration

➢Wire (wire) Definition

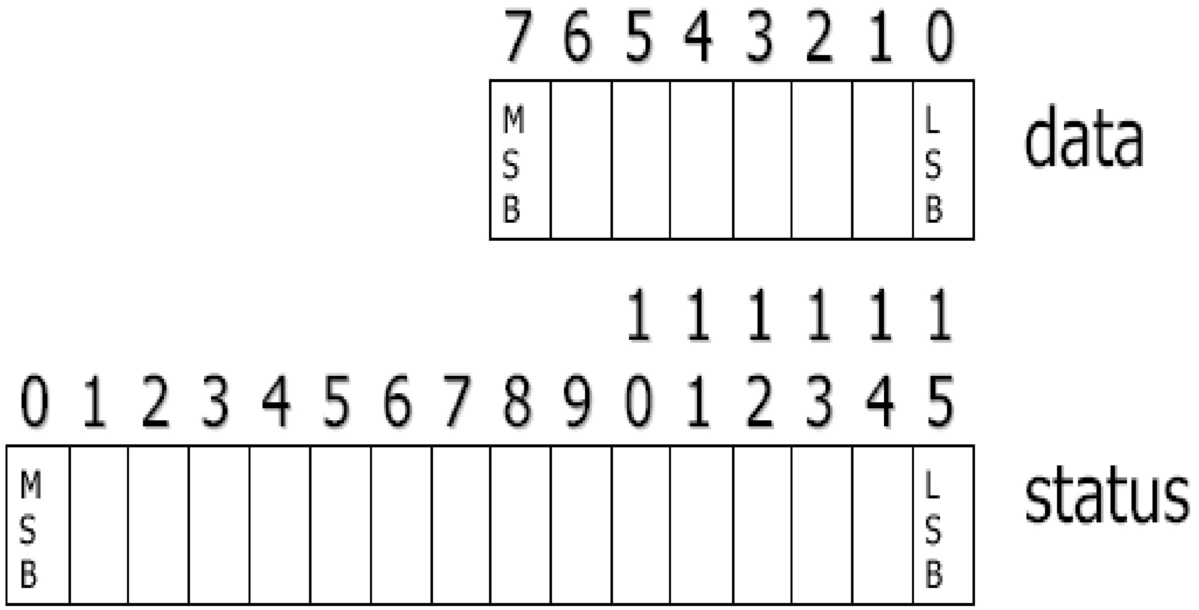
```
wire a, b, c;        // Define 1-bit nets a, b, and c.

wire [7:0]v;         // Define 8-bit wire vector
```



Data Type Declaration

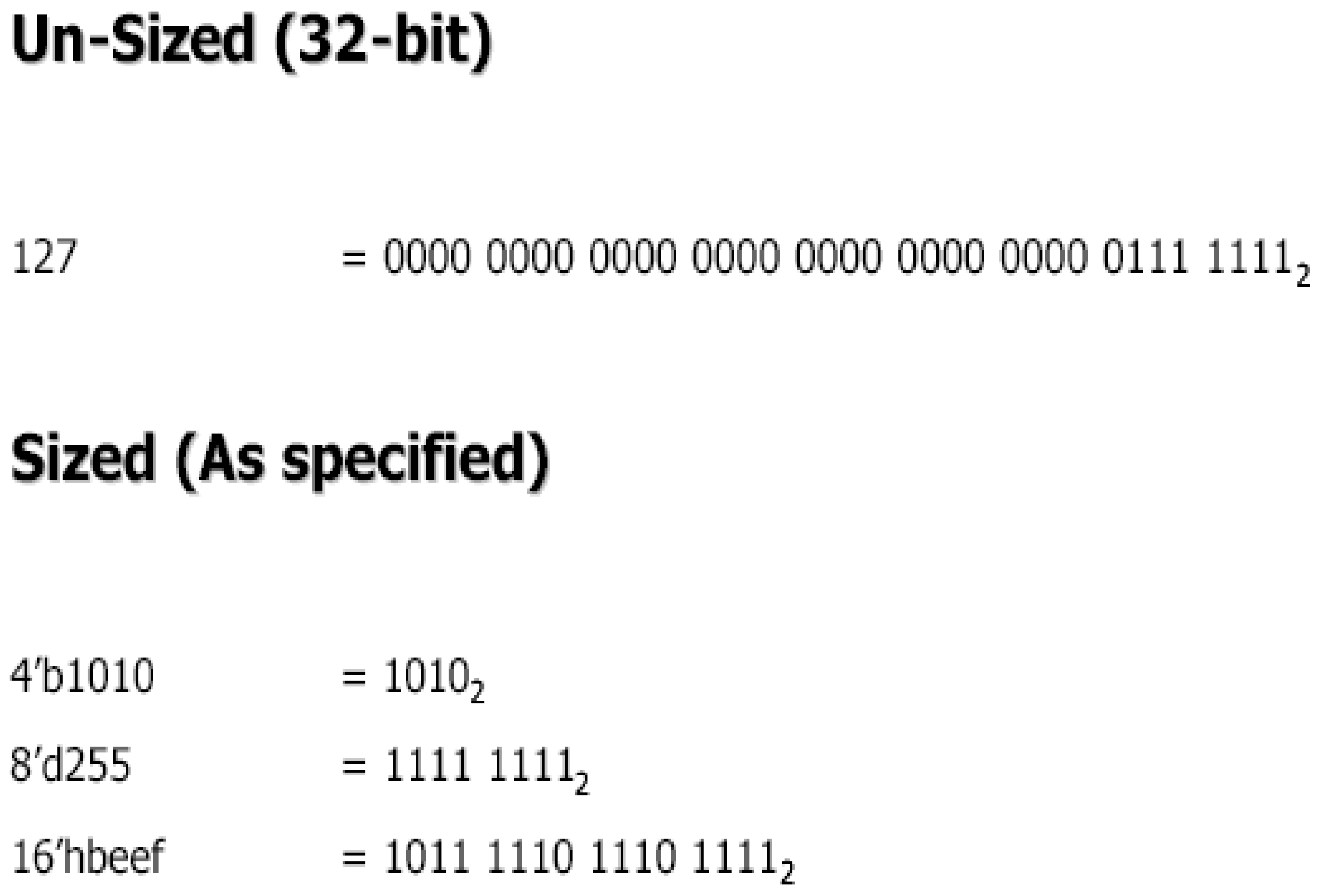
- **Register (reg) Definition**

```
reg [7:0] data;         // 8-bits wide, LSB0

reg [0:15] status;      // 16-bits wide, MSB0
```



Variable Declaration

- constants



Un-Sized (32-bit)

127	= 0000 0000 0000 0000 0000 0000 0000 0111 1111_2

Sized (As specified)

4'b1010	= 1010_2
8'd255	= 1111 1111_2
16'hbeef	= 1011 1110 1110 1111_2

Verilog Arithmetic Operator

+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)
%	(modulus)

```
parameter n = 4;
reg[3:0] a, c, f, g, count;
f = a + c;
g = c - n;
 count = (count +1)%16;            //Can count 0 thru 15.
```

Verilog Relational Operator

<	(less than)
<=	(less than or equal to)
>	(greater than)
>=	(greater than or equal to)
==	(equal to)
!=	(not equal to)

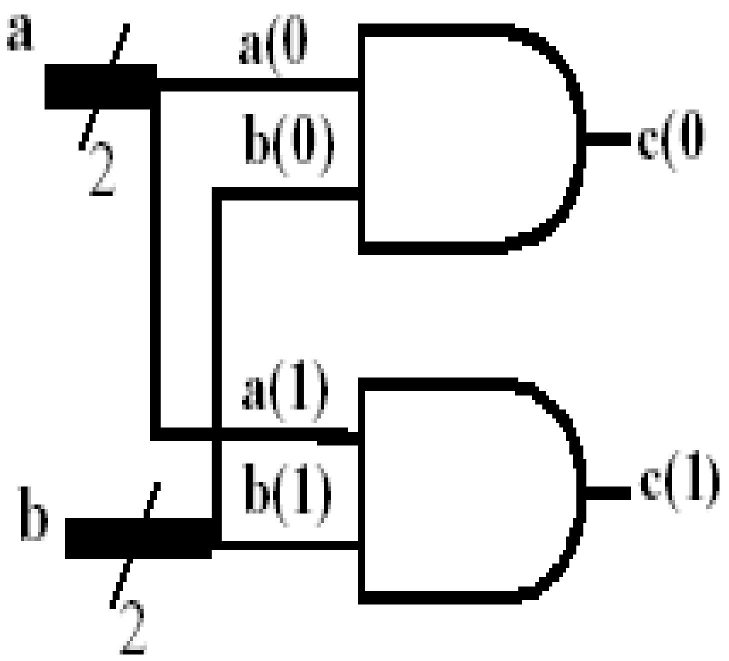
```
if (x = = y)    e = 1;
else            e = 0;
```

```
// Compare in 2's compliment;  a>b
reg [3:0] a,b;
if (a[3]= = b[3])   a[2:0] > b[2:0];
else                b[3];
```

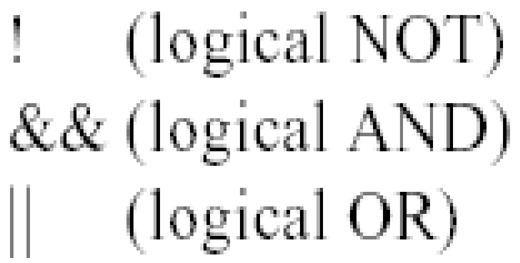
Verilog Bitwise Operator

~	(bitwise NOT)
&	(bitwise AND)
|	(bitwise OR)
^	(bitwise XOR)
~^ or ^~	(bitwise XNOR)

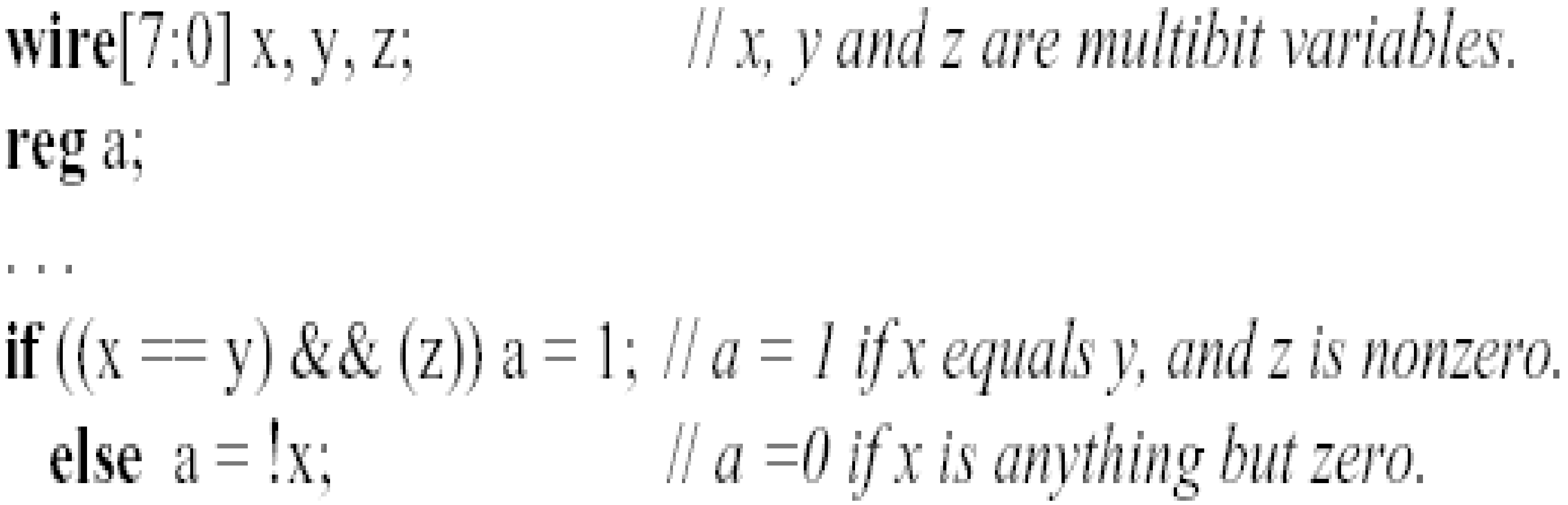
```
module and2 (a, b, c);
  input [1:0] a, b;
  output [1:0] c;
  assign c = a & b;
endmodule
```



Verilog Logical Operator



```
!    (logical NOT)
&&   (logical AND)
||   (logical OR)
```



```
wire[7:0] x, y, z;                  // x, y and z are multibit variables.
reg a;
. . .
if ((x == y) && (z)) a = 1; // a = 1 if x equals y, and z is nonzero.
  else  a = !x;                     // a =0 if x is anything but zero.
```

Verilog Concatenation Operator

{ } (concatenation)

```
wire [1:0] a, b;    wire [2:0] x;      wire [3;0] y, Z;
assign x = {1'b0, a};  // x[2]=0,  x[1]=a[1],  x[0]=a[0]
assign y = {a, b};   /* y[3]=a[1],  y[2]=a[0],  y[1]=b[1],
y[0]=b[0] */

assign {cout, y} = x + Z;  // Concatenation of a result
```

Continuous Assignment: RTL Modeling

- Continuous assignments are made with the assign statement:

assign LHS = RHS;

<u>Rules:</u>

- The left hand side, LHS, must be a wire/reg.
- The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Continuous Assignment

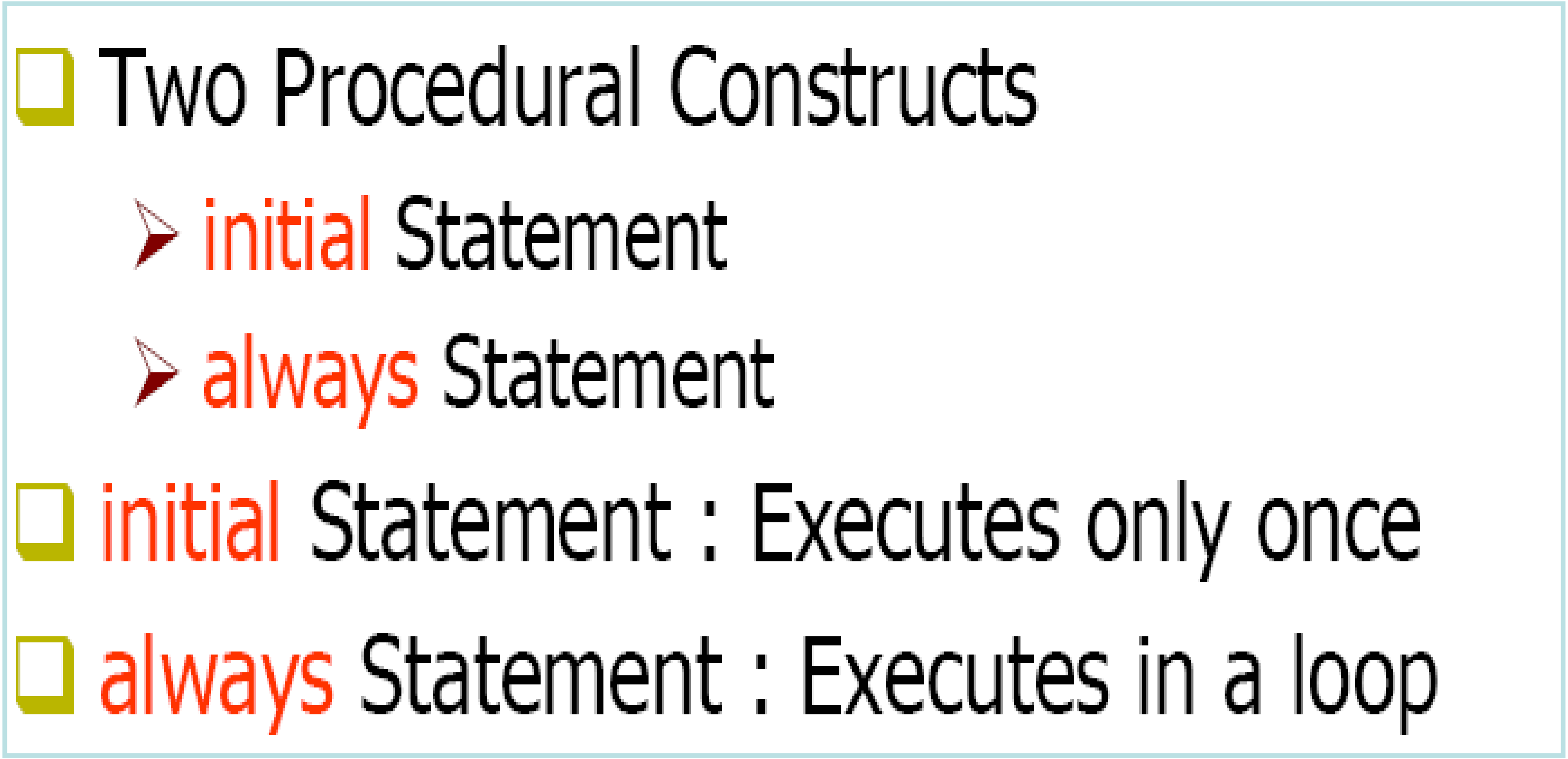
- **Example 1**

```
module two_input_xor (in1, in2, out);
input in1, in2;                       // use these as a wire
output out;                           // use this as a wire
assign out = in1 ^ in2;
endmodule
```

- **Example 2**

```
module two_input_xor (in1, in2, out);
input in1, in2;
output out;
wire product1, product2;
assign product1 = in1 & !in2;      // could have done all in
assign product2 = !in1 & in2;      // assignment of out with
assign out = product1 | product2;  // bigger expression
endmodule
```

Procedural Constructs



- Two Procedural Constructs
  - initial Statement
  - always Statement
- initial Statement : Executes only once
- always Statement : Executes in a loop

Syntax Example

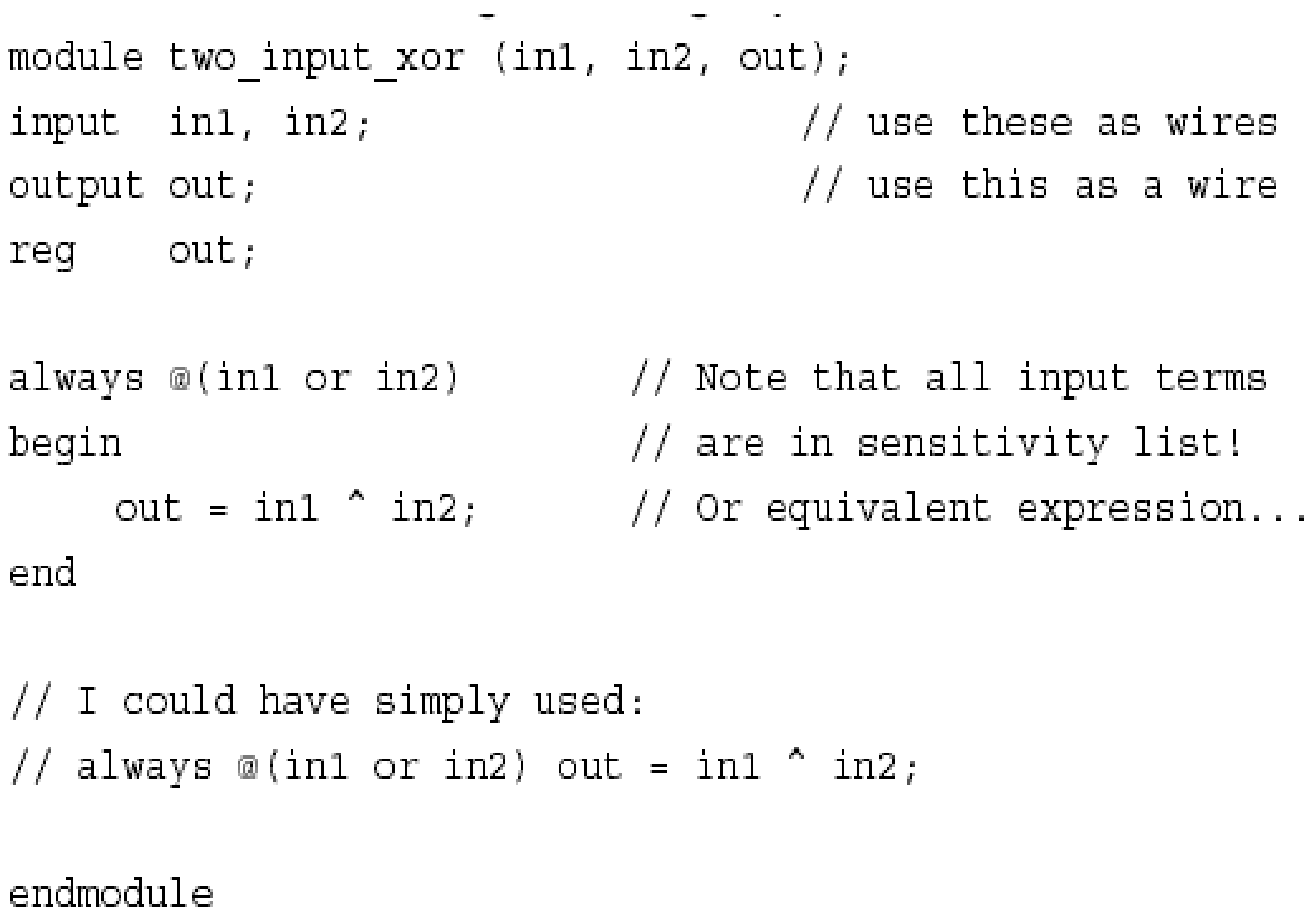
```
initial
begin
    // These procedural assignments are executed
    // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
    // These procedural assignments are executed
    // whenever the events in the sensitivity list
    // occur.
end
```

Sensitivity List

- always @(a or b) // any changes in a or b
- always @(posedge a) // a transitions from 0 to 1
- always @(negedge a) // a transitions from 1 to 0
- always @(a or b or negedge c or posedge d)

Procedural Constructs

Combinational logic using operators:



```
module two_input_xor (in1, in2, out);
input  in1, in2;                        // use these as wires
output out;                             // use this as a wire
reg    out;

always @(in1 or in2)         // Note that all input terms
begin                        // are in sensitivity list!
    out = in1 ^ in2;         // Or equivalent expression...
end

// I could have simply used:
// always @(in1 or in2) out = in1 ^ in2;

endmodule
```

Procedural Constructs

Combinational logic using if-else:

```
module two_input_xor (in1, in2, out);
input  in1, in2;                   // use these as wires
output out;                        // use this as a wire
reg    out;

always @(in1 or in2)       // Note that all input terms
begin                      // are in sensitivity list!
     if (in1 == in2) out = 1'b0;
     else out = 1'b1;
end

endmodule
```

Procedural Constructs

Combinational logic using case:

```
module two_input_xor (in1, in2, out);
input  in1, in2;                         // use these as wires
output out;                              // use this as a wire
reg    out;

always @(in1 or in2)          // Note that all input terms
begin                         // are in sensitivity list!
     case ({in2, in1})        // Concatenated 2-bit selector
     2'b01:  out = 1'b1;
     2'b10:  out = 1'b1;
     default: out = 1'b0;
     endcase
end
endmodule
```

System Tasks

- The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  - $display(“The value of a is %b”, a);
    - Used in procedural blocks for text output.
    - The %b is the value format (binary, in this case…)
  - $finish;
    - Used to finish the simulation.
    - Use when your stimulus and response testing is done.

Event Control

- Event Control
  - Edge Triggered Event Control
  - Level Triggered Event Control
- Edge triggered Event Control

```
@ (posedge CLK) //Positive Edge of CLK
  Curr_State = Next_state;
```

@ negedge	@ posedge
$1 \to x$	$0 \to x$
$1 \to z$	$0 \to z$
$1 \to 0$	$0 \to 1$
$x \to 0$	$x \to 1$
$z \to 0$	$z \to 1$

- Level Triggered Event Control

```
@ (A or B) //change in values of A or B
  Out = A & B;
```

Loop Statement

- Loop Statement
  - Repeat
  - While
  - For
- Repeat Loop
  - Example

```
repeat (count)
    sum = sum + 6;
```

  - If condition is a x or z is treated as o

Loop Statement (cont.)

While Loop

- Example:

```
while (Count < 10) begin
  sum = sum + 5;
  Count = Count +1;
end
```

- If condition is a x or z it is treated as 0

For Loop

- Example:

```
for (Count = 0; Count < 10; Count = Count + 1) begin
  sum = sum + 5;
end
```

Conditional statement

- if Statement
- Format:

```
if (condition)
procedural_statement
else if ( condition)
procedural_statement
```

- Example

```
if (Clk)
   Q = 0;
else
   Q = D;
```

Case Statement

Example 1:

```
case (X)
  2'b00: Y = A + B;
  2'b01: Y = A – B;
  2'b10: Y = A / B;
endcase
```

Example 2:

```
case (3'b101 << 2)
  3'b100: A = B + C;
  4'b0100: A = B – C;
  5'b10100: A = B / C; //This statement is executed
endcase
```

Structural

```
module mux21(a, b, s, y);
   input a, b, s;
   output y;
   wire m, n, p;
   and g1(m, b, s);
   not g2(n, s);
   and g3(p, a, n);
   or g4(y, m, p);
endmodule
```

RTL

```
module mux21(a, b, s, y);
  input a, b, s;
  output y;
  assign y = s ? b : a;
endmodule
```

Behavioral

```
module mux21(A, B, S, Y);
    input A, B, S;
    output Y;
    reg Y;
    always @(A or B or S)begin
            if(S==0) Y = A;
            else Y = B;
    end
endmodule
```

Compiler Directives

‘include – used to include another file

- ➢ Example

‘include “./pqp_fetch.v”

`define – (Similar to #define in C) used to define global parameter

Example:

```
 `define BUS_WIDTH 16
 reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
```

`undef – Removes the previously defined directive

Example:

```
 `define BUS_WIDTH 16
   …
   reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
   …
 `undef BUS_WIDTH
```

Suggested Coding Style

- Write one module per file, and name the file the same as the module. Break larger designs into modules on meaningful boundaries.
- Always use formal port mapping of sub-modules.
- Use parameters for commonly used constants.
- Be careful to create correct sensitivity lists.

- Don’t ever just sit down and “code”. Think about what hardware you want to build, how to describe it, and how you should test it.
- You are not writing a computer program, you are describing hardware… ***Verilog is not C!***
- Only you know what is in your head. If you need help from others, you need to be able to explain your design -- either verbally, or by detailed comments in your code.

The End