



# **DIGITAL SYSTEM DESIGN**

## **Lecture 3**

### **Verilog HDL**

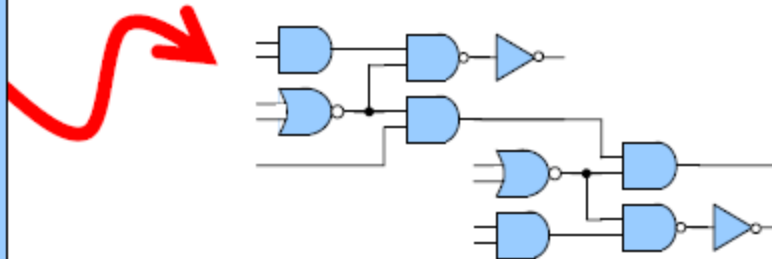
**Waqar Ahmad**

**Faculty of Computer Science and Engineering**

# HDL VS SOFTWARE PROGRAMMING LANGUAGES

- Software Programming Language:
  - Language that can be translated into machine instructions and then executed on a computer
- Hardware Description Language:
  - A language used to describe a digital system, for example, a microprocessor or a memory or a simple flip-flop. This just means that, by using a HDL one can describe any hardware (digital ) at any level.

```
module foo(clk,xi,yi,done);  
  input [15:0] xi,yi;  
  output done;  
  
  always @(posedge clk)  
  begin:  
    if (!done) begin  
      if (x == y) cd <= x;  
      else (x > y) x <= x - y;  
    end  
  end  
endmodule
```



# WHY HDLS?

Digital Design

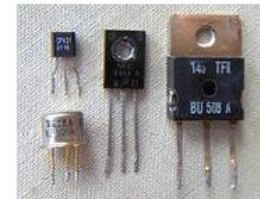
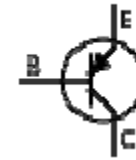
Let's Build an AND gate

Draw Schematic

Select Components

Implement

Idea



# WHY HDLs?

Digital Design

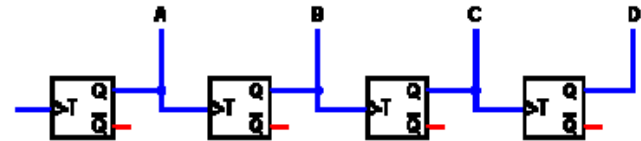
Let's Build a counter

Draw Schematic

Select Components

Implement

Idea



# WHY HDLs?

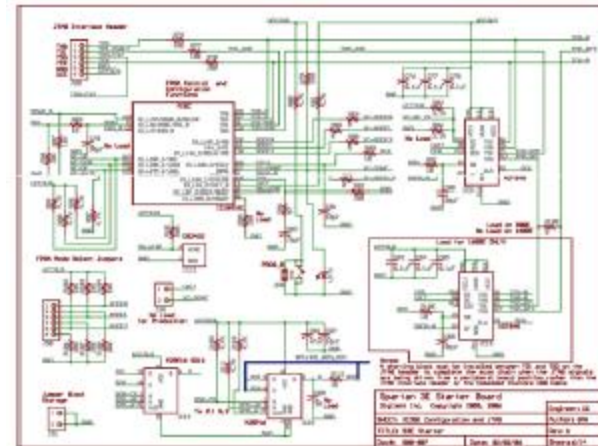
Digital Design

Let's Build an iPOD

Draw Schematic

Duh??

Idea

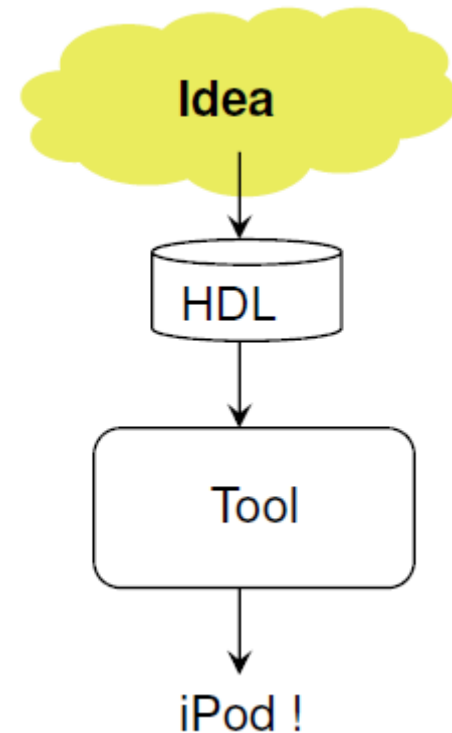


Configuration Logic  
(You haven't even started on  
iPOD)



# WHY HDL

- Basic idea is a programming language to describe hardware
- Initial purpose was to allow abstract design and simulation
  - Design could be verified then implemented in hardware
- Now Synthesis tools allow direct implementation from HDL code.
  - Large improvement in designer productivity



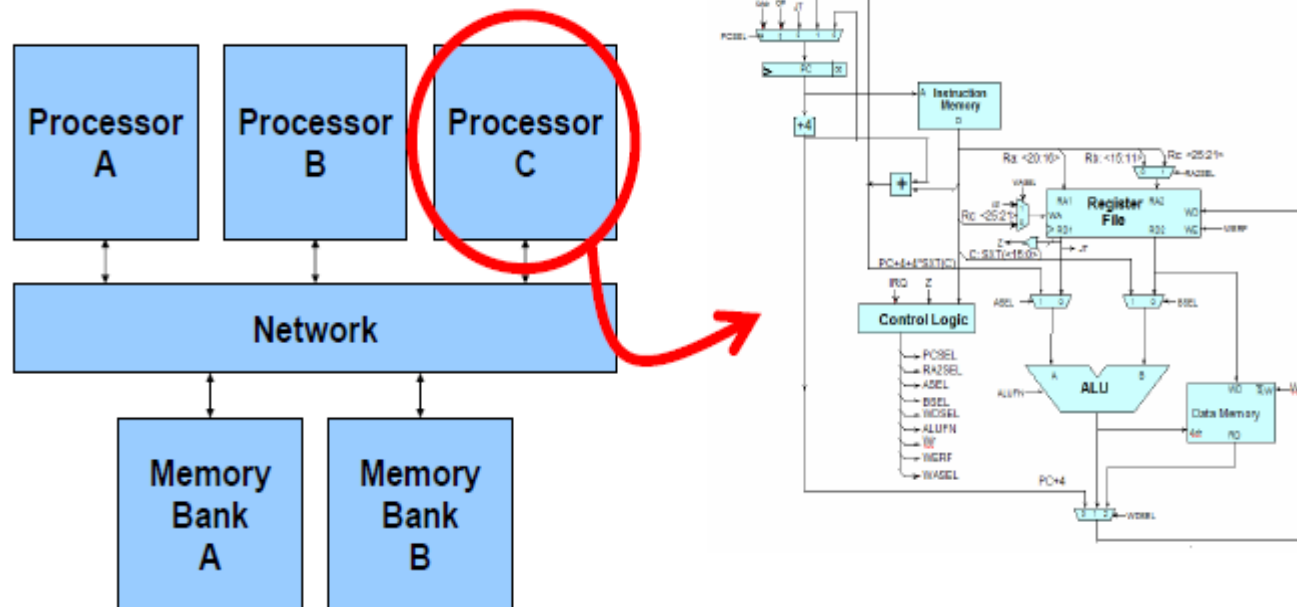
# ADVANTAGES OF HDL

- HDL allows write-run-debug cycle for hardware development.
  - Similar to programming software
  - Much, much faster than design-implement-debug
- Combined with modern Field Programmable Gate Array chips large complex circuits (100000s of gates) can be implemented.



# ADVANTAGES OF HDL

- Allows designers to talk about what the hardware should do without actually designing the hardware itself.
- In other words HDLs allow designers to **separate behavior from implementation** at various levels of abstraction





# HDLs

- There are many different HDLs
  - Verilog HDL
  - ABEL
  - VHDL
- We will use Verilog HDL



# VERILOG HDL

- Verilog HDL is second most common
  - Easier to use in many ways
  - C - like syntax
- History
  - Developed as proprietary language in 1985
  - Opened as public domain spec in 1990
    - Due to losing market share to VHDL
  - Became IEEE standard in 1995



# LANGUAGE RULES

- Verilog is a case sensitive language (with a few exceptions)
- First character not a digit
- Identifiers (space-free sequence of symbols)
  - upper and lower case letters from the alphabet
  - digits (0, 1, ..., 9)
  - underscore ( \_ )
  - \$ symbol (only for system tasks and functions)
- Terminate lines with semicolon
- Single line comments: // A single-line comment goes here



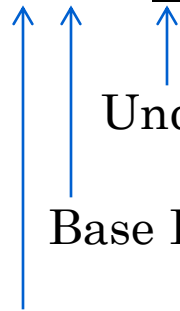
# NUMBER REPRESENTATION

## ○ Data Values

- 0 – Logic 0
- 1 – Logic 1
- X – Unknown/Don't Care
- Z – High Impedance

## ○ Numeric Literals

- 4'b10\_11



Underscores are ignored

Base Format (b, d, o, h)

Decimal Number Indicating Size in bits

e.g. 32'h35A2\_6F7X

In Verilog default is decimal i.e. 13 = 'd13



## ○ Examples:

- `6'b010_111` gives `010111`
- `8'b0110` gives `00000110`
- `8'b1110` gives `00001110`
- `4'bx01` gives `xx01`
- `16'H3AB` gives `0000001110101011`
- `24` gives `0...0011000`
- `5'O36` gives `11100`
- `16'Hx` gives `xxxxxxxxxxxxxxxxxxxx`
- `8'hz` gives `zzzzzzzz`



# DESIGN ENCAPSULATION

- Encapsulate structural and functional details in a module

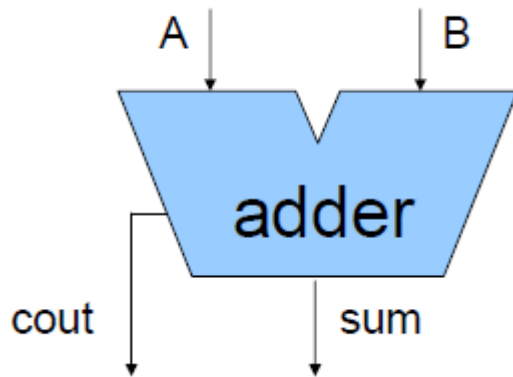
```
module my_design (port_list);  
    ... // Declarations of ports go here  
    ... // Structural and functional details go here  
endmodule
```

- Encapsulation makes the model available for **instantiation** in other modules
- Ports can be **input, output, inout**
  - Ports are similar to pins on chip.



# PORTS DECLARATION

- Must specify direction and width of each port



```
module adder(cout, sum, A, B);
```

```
input [3:0] A, B;
```

```
output cout;
```

```
output [3:0] sum;
```

```
// HDL modeling of  
// adder functionality
```

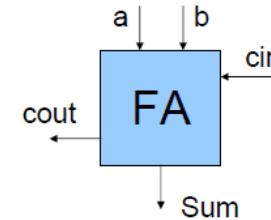
```
endmodule
```

Don't forget to add semi-colon at the end of each line **except** "endmodule"

# MODULE INSTANTIATION

- A module can contain other modules through **module instantiation** creating a module hierarchy.
  - Modules are connected together with nets
  - Ports are attached to nets either by position or by name

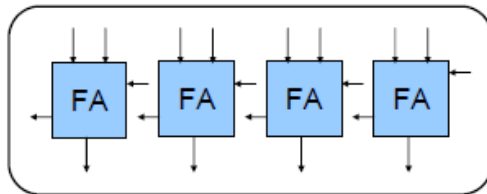
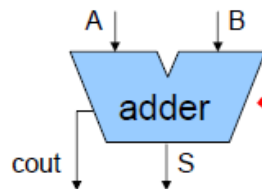
```
module FA(cout, sum, a, b, cin);  
    input a, b, cin;  
    output cout, sum;  
    // HDL modeling of 1 bit  
    // adder functionality  
endmodule
```



```
module adder_4bit (cout, sum, A, B);  
    input [3:0] A, B;  
    output cout;  
    output [3:0] sum;  
  
    FA fa0( ... );  
    FA fa1( ... );  
    FA fa2( ... );  
    FA fa3( ... );  
  
endmodule
```

Type  
Name

Instance  
Name



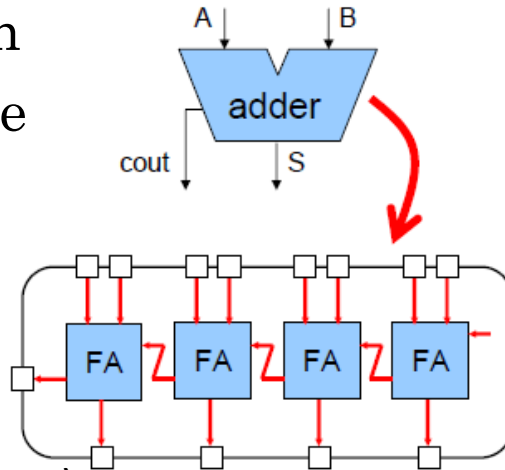
endmodule



# HIERARCHICAL MODELING

## ○ Ports are attached to nets either

- by position
- or by name



```
module adder(cout, S, A, B);
```

```
  input [3:0] A, B,
```

```
  output cout,
```

```
  output [3:0] S
```

```
  wire c0, c1, c2;
```

```
  //by name
```

```
  FA fa0( .a(A[0]), .b(B[0]), .cin(0), .cout(c0), .sum(S[0]) );
```

```
  FA fa1( .a(A[1]), .b(B[1]), ...
```

```
  ....
```

```
endmodule
```

```
module adder(cout, S, A, B);
```

```
  input [3:0] A, B,
```

```
  output cout,
```

```
  output [3:0] S
```

```
  wire c0, c1, c2;
```

```
  //by position
```

```
  FA fa0( A[0], B[0], 0, c0, S[0] );
```

```
  FA fa1( A[1], B[1], c0, c1, S[1] );
```

```
  FA fa2( A[2], B[2], c1, c2, S[2] );
```

```
  FA fa3( A[3], B[3], c2, cout, S[3] );
```

```
endmodule
```

# ABSTRACTION LEVELS

## ○ Structural

- Textual replacement for schematic
- Hierarchical composition of modules from primitives
- Instantiation of primitives and modules
- Gate Level Modeling

## ○ Behavior

- Describe **what** module does, not how
- Synthesis generates circuit for module
- Dataflow Modeling
  - continuous assignments
- Behavioral Modeling
  - procedural assignments



# GATE LEVEL MODELING

- A logic circuit can be designed by use of logic gates.
- Verilog supports basic logic gates as predefined *primitives*.
- These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.
- The built-in gates include *and*, *nand*, *or*, *nor*, *xor*, *xnor*, *buf*, *not*, *bufif0*, *bufif1*, *notif0*, *notif1*, *pullup* and *pulldown* gates.

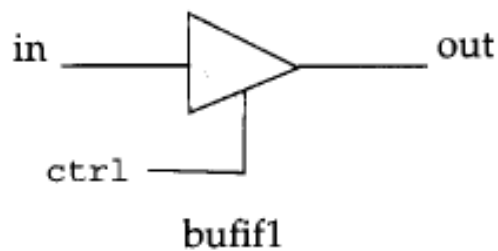


# THREE-STATE GATES

- Three-state gates have a control input that can place the gate into a high-impedance state. (symbolized by **z** in HDL).
- The **bufif1** gate behaves like a normal buffer if *control=1*. The output goes to a high-impedance state **z** when *control=0*.
- **bufif0** gate behaves in a similar way except that the high-impedance state occurs when *control=1*
- Two **not** gates operate in a similar manner except that the o/p is the complement of the input when the gate is not in a high impedance state.
- The gates are instantiated with the statement
  - `gate_name instance_name (output, input, control);`

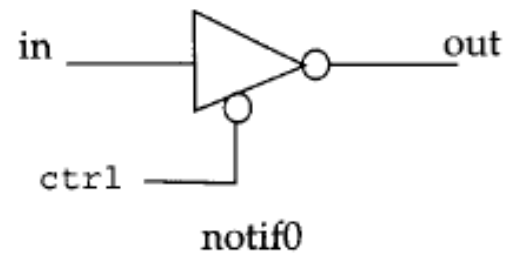


# bufif / notif GATES



		ctrl			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
		0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x



# GATE LEVEL MODELING (SIMPLE EXAMPLE)

```
module simplecct(out, sel, a, b);
```

```
  input a,b,sel;
```

```
  output out,
```

```
  wire sel_neg, sel_a, sel_b;
```

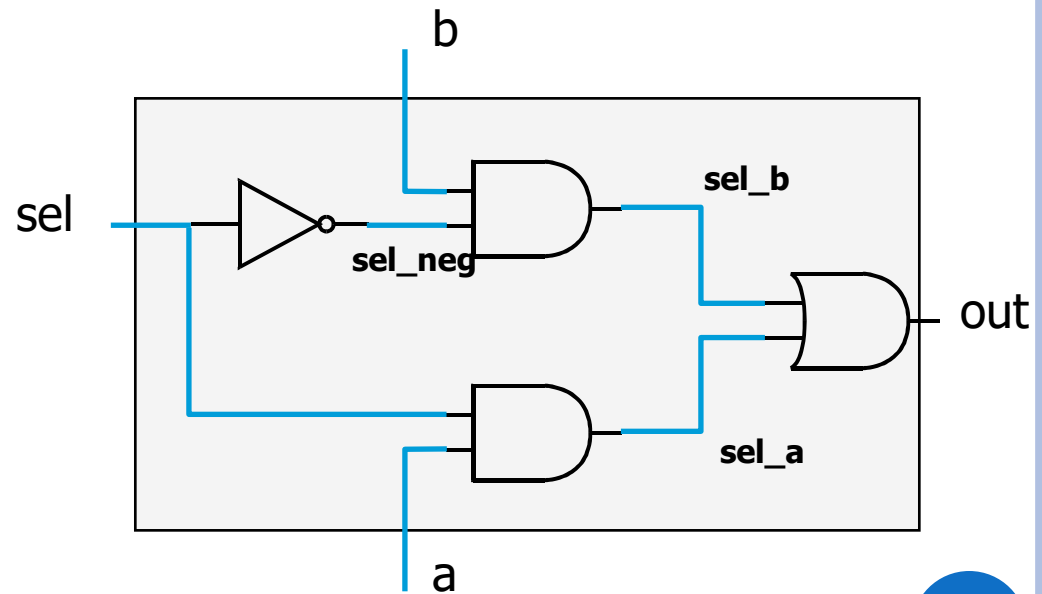
```
  not n1(sel_neg, sel);
```

```
  and a1(sel_b, b, sel_neg);
```

```
  and a2(sel_a, a, sel);
```

```
  or o1(out, sel_b, sel_a);
```

```
endmodule
```



# GATE LEVEL MODELING (EXAMPLE)

```
module mux4x1(out, a, b, c, d, sel);
```

```
    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 );
```

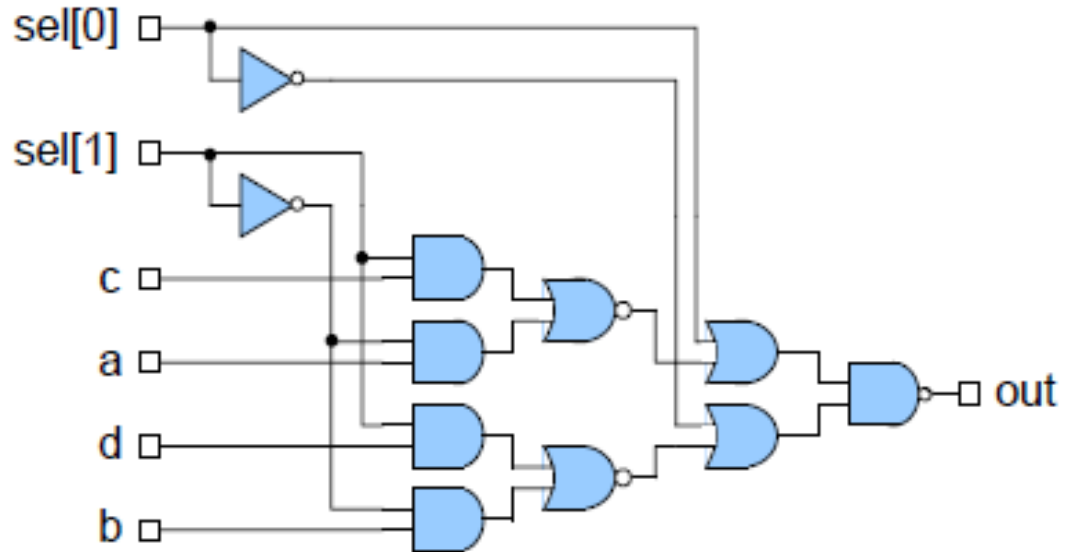
```
    wire y0, y1;
```

```
    or or0( y0, x0, sel[0] );
```

```
    or or1( y1, x1, sel_b[0] );
```

```
    nand nand0( out, y0, y1 );
```

```
endmodule
```



What is this circuit???

Verify!!!



# GATE LEVEL MODELING (DELAYS)

- Default gate delay is **zero** in Verilog.
- Sometimes it is necessary to specify the amount of delay from the input to the output of gates.
- In Verilog, the delay is specified in terms of time units and the symbol #.
- The association of a time unit with physical time is made using ***timescale*** compiler directive.
- Compiler directive starts with the “backquote (”)” symbol.
  - ``timescale 1ns/100ps`
    - First number specifies the *unit of measurement* for time delays.
    - Second number specifies the *precision* for which the delays are rounded off, in this case to 0.1ns.

- Allowed unit/precision values:

{1 | 10 | 100, s | ms | us | ns | ps}

- Example:

```
`timescale 10ps / 1ps
```

```
nor #3.57 nor1(z, x1, x2); //nor delay used = 3.57 x 10 ps = 35.7 ps => 36 ps
```





# CIRCUIT WITH DELAY (EFFECTS)

```
module cwd(A,B,C,x,y);  
  input A,B,C;  
  output x,y;  
  wire e;  
  and #(30) g1(e,A,B);  
  or #(20) g3(x,e,y);  
  not #(10) g2(y,C);  
endmodule
```

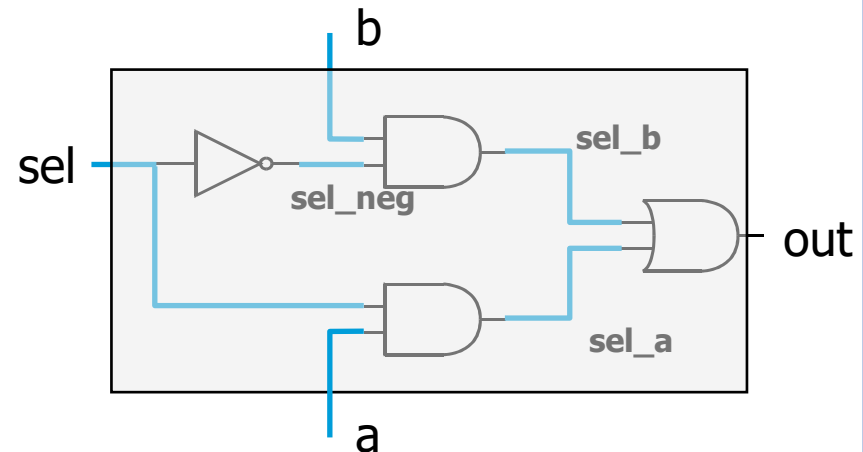
Time (ns)	Input A B C	Output y e x
<0	0 0 0	1 0 1
0	1 1 1	1 0 1
10	1 1 1	0 0 1
20	1 1 1	0 0 1
30	1 1 1	0 1 0
40	1 1 1	0 1 0
50	1 1 1	0 1 1



# DATAFLOW MODELING

- Specify output signals in terms of input signals.
- Keyword: **assign**

```
module simplecct(out, sel, a, b);  
  input a,b,sel;  
  output out;  
  
  assign out = (sel & a) | (~sel & b);  
endmodule
```



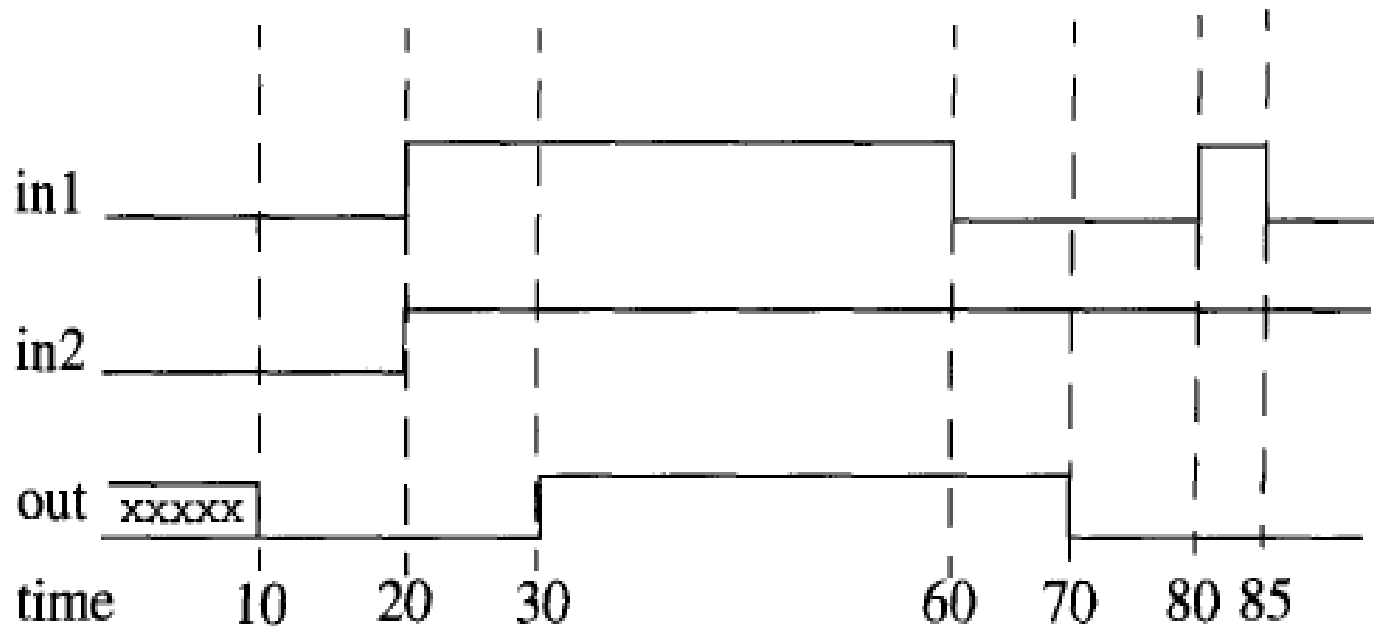
- This is called a **continuous assignment** since the RHS is always being evaluated and the result is continuously being driven onto the net on the LHS. (always active)
- The left hand side of an assignment must always be a scalar or vector net. It cannot be a scalar or vector register.



# ADDING DELAY

- Delay values can be specified for assignments in terms of time units. Default delay is **zero**.
- Delay values are used to control the time when a net is assigned the evaluated value

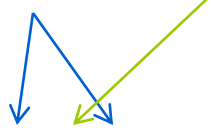
`assign #10 out = in1 & in2; //Delay in continuous assign`



# OPERATORS AND OPERANDS

- Dataflow modeling uses a number of operators that act on operands to produce desired results.
- Verilog HDL provides about 30 operator types.
- The value assigned to the net is specified by an expression that uses operands and operators.

assign {c0, d} = a + b;



# OPERAND TYPES

assign {c0, d} = a + b;

Left-Hand Side = Right-Hand Side

nets (wire)

nets (wire)

variable (reg)

parameters

numbers

function call



# OPERATOR TYPES

- **Arithmetic operators:**  $+$  (ADD),  $-$  (SUB),  $*$  (MUL),  $/$  (DIV),  $\%$  (MOD)
  - Number of operands: 2
  - Divide ( $/$ ) and modulus ( $\%$ ) operators are synthesizable only if the right side operand is a power of 2.
- **Binary Bitwise:**  $\sim$  (NOT),  $\&$  (AND),  $\sim\&$  (NAND),  $|$  (OR),  $\sim|$  (NOR),  $\wedge$  (XOR),  $\sim\wedge$  (XNOR)
  - Number of operands: 2 (For **NOT = 1**)
  - Perform a bit-by-bit comparison.
  - If one operand is shorter than the other, the shorter operand is bit-extended with zeroes to match length
  - Examples:
    - $4'b0100 | 4'b1001 = 4'b1101$
    - $\sim 8'b0110\_1100 = 8'b1001\_0011$



# OPERATOR TYPES

## ○ Unary Reduction: $\&$ (AND), $\sim\&$ (NAND), $|$ (OR), $\sim|$ (NOR), $\wedge$ (XOR), $\sim\wedge$ (XNOR)

- Condense all bits from a vector into a single bit using the specified logical operation. Number of operands = 1
- Yield a 1-bit result: 0 or 1
- Example:

$\& 4'b0101 = 0 \& 1 \& 0 \& 1 = 0$

$| 4'b0101 = 0 | 1 | 0 | 1 = 1$

$\wedge 4'b0111 = 0 \wedge 1 \wedge 1 \wedge 1 = 1$

## ○ Logical Operators: $!$ (Logical NOT), $\&\&$ (Logical AND), $||$ (Logical OR)

- Number of operands: NOT: 1 / AND, OR = 2
- For bit vector inputs, first reduce-OR operands, then perform a bitwise or/and/not corresponding to logical operation

$4'b0000 || 4'b0111 = 1$

$4'b0000 \&\& 4'b0111 = 0$

$4'b1100 \&\& 4'b0011 = 1$

$! 4'b0000 = 1$



# OPERATOR TYPES

- **Relational Operators:**  $<$  (less than),  $>$  (greater than),  $<=$  (less than or equal),  $>=$  (greater than or equal),  $==$  (equality),  $!=$  (inequality)

- Number of operands: 2
- Always yield a logical value: 0 or 1
- Examples:

$(4'b1011 < 4'b0111) = 0$

$(4'b1011 != 4'b0111) = 1$

- **Shift Operators:**  $<<$  (Logical shift-left),  $>>$  (Logical shift-right)

- Example:

$4'b1110 >> 2 \rightarrow 0011$





# OPERATOR TYPES

## ○ Replication Operator:

- We can replicate things together
- Example:

$A = 2'b01;$                        $B = \{3\{A\}\}; // B = 010101$

## ○ Concatenation Operator: { , }

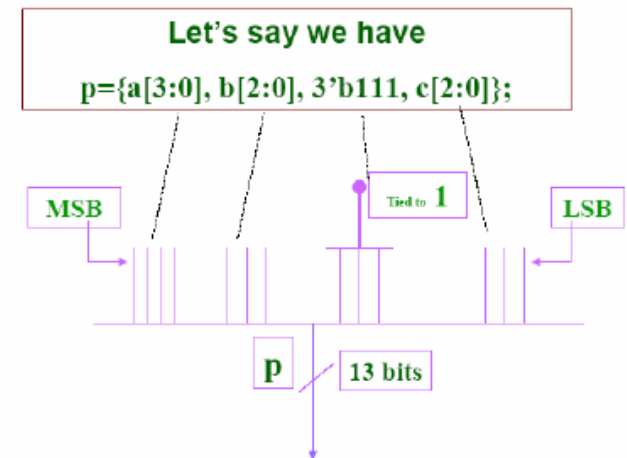
- Appends multiple operands
- Example:

$\{5'b10110, 2'b10, 1'b0\} = 8'b10110100$

$x = \{a, b[3:2], c\} = \{a, b[3], b[2], c\}; // x$  is a 4-bit vector

- Concatenation can be replicated

$\{a, 3\{b, c\}\} = \{a, b, c, b, c, b, c\};$



# OPERATOR TYPES

## ○ Conditional Operator: ? :

- *condition ? if\_true-expression : if\_false-expression;*
  - The **condition** is first evaluated
  - If the result is true (logic 1): **if true-expression** is evaluated
  - If the result is false (logic 0): **if false-expression** is evaluated
- Example:

```
//Dataflow description of 2-to-1-line mux  
module mux2x1_df (Out, A, B, select);  
    input A, B, select;  
    output Out;  
    assign Out = select ? A : B;  
endmodule
```



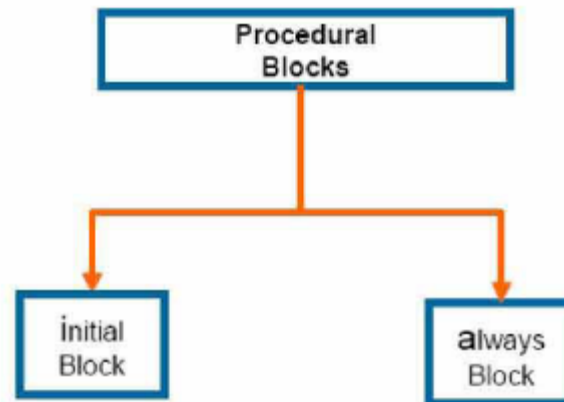
# CLASS TASKS

- Write Verilog codes of
  - a 4x1 Mux using
    - Logic Equation
    - Conditional Operator (Nested)
  - 4-bit Adder



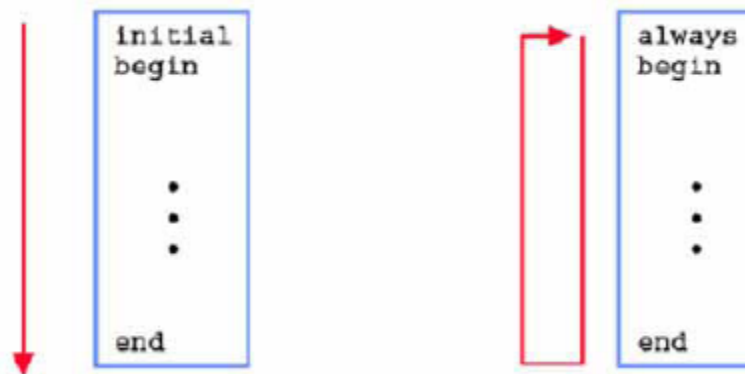
# BEHAVIORAL MODELING

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- In Behavioral Modeling, everything comes in a procedural block.
- There are two kinds of procedural blocks in Verilog HDL: **initial** and **always**.



# PROCEDURAL BLOCKS

- The **initial** behavior executes once beginning at time=0.
- The **always** behavior executes and re-executes repeatedly (whenever the given condition is true) until the simulation terminates.
  - The statements within the block, after the event control expression, execute sequentially and the execution suspends after the last statement has executed.
  - Then the always statement waits again for an event to occur.



# BEHAVIORAL MODELING

- A behavior is declared within a module by using the keywords **initial** or **always**, followed by a statement or a block of statements enclosed by the keywords **begin** and **end**.
- The target output of procedural assignment statements must be of the **reg** data type.
- A **reg** data type retains its value until a new value is assigned.



# **always** BLOCK

- **always** statement : Sequential Block
- Sequential Block: All statements within the block are executed sequentially
- When is it executed?
  - Occurrence of an event in the sensitivity list
  - Event: Change in the logical value
- Statements with a Sequential Block: Procedural Assignments
- Delay in Procedural Assignments
  - Inter-Statement Delay
  - Intra-Statement Delay



# EVENTS IN SENSITIVITY LIST

- Event Control
  - Level Triggered Event Control
  - Edge Triggered Event Control
- Level sensitive (e.g. in combinational circuits and latches)

`always @ (A or B or reset) //change in values of A or B`

Sensitivity  
List

- will cause the execution of the procedural statements in the **always** block if changes occur in **A** or **B** or **Reset**.

- Edge sensitive:

`always @ (posedge Clk or negedge reset) //Positive Edge of CLK`

- will cause the execution of the procedural statements only at the rising edge of **clock** or falling edge of the **reset**.





# BEHAVIORAL MODELING (EXAMPLE)

//Behavioral description of 2-to-1-line multiplexer

```
module mux2x1_bh(OUT,A,B,select);  
    input A,B,select;  
    output OUT;  
    reg OUT;  
  
    always @ (select or A or B)  
        begin  
            if      (select == 1)      OUT = A;  
            else                                OUT = B;  
        end  
  
endmodule
```

- Note that there is no “;” at the end of always statement



# 4-INPUT MULTIPLEXER

```
module mux4x1(Out,A,B,C,D,select);  
    input A,B,C,D;  
    input [1:0] select;  
    output Out;  
    reg Out;  
    always @ (select or A or B or C or D)  
        begin  
            if (select == 0)  
                Out = A;  
            else if (select == 1)  
                Out = B;  
            else if (select == 2)  
                Out = C;  
            else //if (select == 3)  
                Out = D;  
        end  
endmodule
```

What happens if we accidentally  
leave off a signal on the  
sensitivity list?

The always block will not  
execute if just D changes – so  
if (select == 3) and d changes,  
Out will not be updated



# 4-INPUT MULTIPLEXER

```
module mux4x1(Out,A,B,C,D,select);  
    input A,B,C,D;  
    input [1:0] select;  
    output Out;  
    reg Out;  
    always @ (*)  
        begin  
            if (select == 0)  
                Out = A;  
            else if (select == 1)  
                Out = B;  
            else if (select == 2)  
                Out = C;  
            else //if (select == 3)  
                Out = D;  
        end  
endmodule
```

In Verilog 2001, we can use the construct @ (\*), which automatically creates a sensitivity list for all the signals read in the always block



# PROCEDURAL ASSIGNMENTS

- A procedural assignment is an assignment within an **initial** or **always** statement.
- There are two kinds of procedural assignments:
  - Blocking assignments
    - executed sequentially in the order they are listed in a sequential block
    - keyword: **=**  
     $B = A;$   
     $C = B + 1;$
  - Non-blocking assignments
    - evaluate the expressions on the right hand side, but do not make the assignment to the left hand side until all expressions are evaluated.
    - keyword: **<=**  
     $B <= A;$   
     $C <= B + 1;$



# BLOCKING ASSIGNMENTS

- Sequence of blocking assignments executes sequentially

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

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

```
begin
```

```
    a = b + 2;
```

```
    b = a * 3;
```

```
end
```

1. Read value of b and add 2 to it;
2. Assign the result to a;
3. Read the value of a and multiply by three;
4. Assign the result to b;

- assignment on *a* MUST complete before assignment on *b* can start



# NON-BLOCKING ASSIGNMENTS

- Sequence of non-blocking assignments executes concurrently

`always @ (posedge clk)`

`begin`

`b <= a + a;`

`c <= b + a;`

`d <= c + a;`

`end`

1. Calculate  $2a$ ,  $b + a$ ,  $c + a$ ;
2. Assign  $b = 2a$ ,  $c = b + a$ ,  $d = c + a$ ;  
(The values calculated in step 1)



# BLOCKING VS NON-BLOCKING

## BLOCKING

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

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a = b + 2;
```

```
    b = a * 3;
```

```
end
```

$a = 0 + 2 = 2;$

$b = 2 * 3 = 6;$

## NON-BLOCKING

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

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a <= b + 2;
```

```
    b <= a * 3;
```

```
end
```

$a = 0 + 2 = 2;$

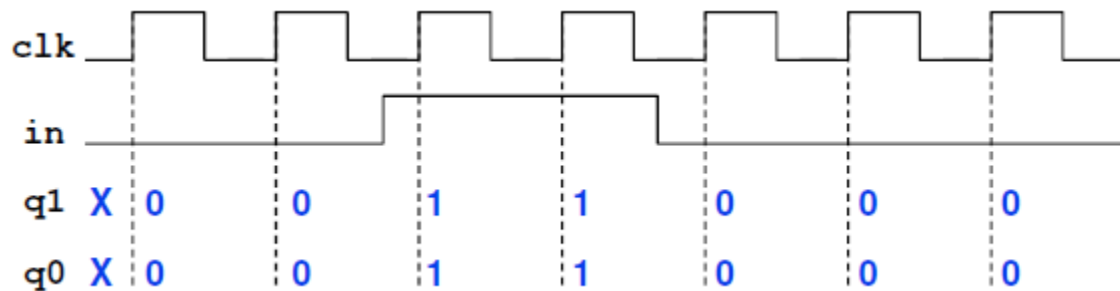
$b = 4 * 3 = 12;$



# ANOTHER EXAMPLE

Using blocking assignments:

```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 = in;  
        q0 = in | q1;  
    end  
endmodule
```

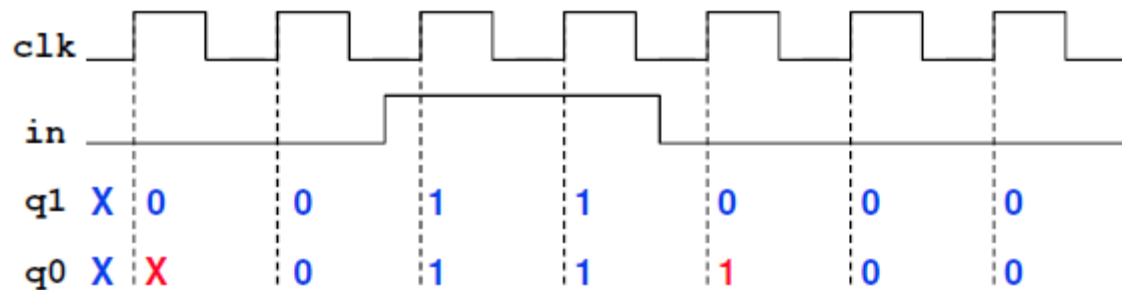




# ANOTHER EXAMPLE

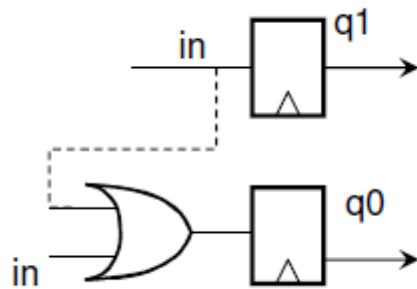
Using non-blocking assignments:

```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 <= in;  
        q0 <= in | q1;  
    end  
endmodule
```

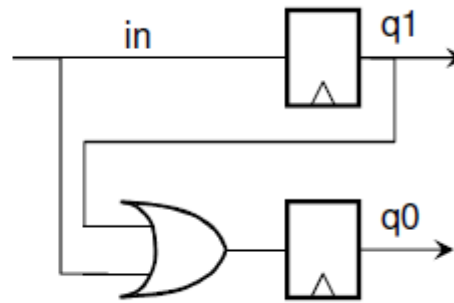


# WHAT'S THE DIFFERENCE???

```
module fsm_mod(q1, q0, in, clk);  
  output q1, q0;  
  input clk, in;  
  reg q1, q0;  
  
  always @(posedge clk) begin  
    q1 = in;  
    q0 = in | q1;  
  end  
endmodule
```

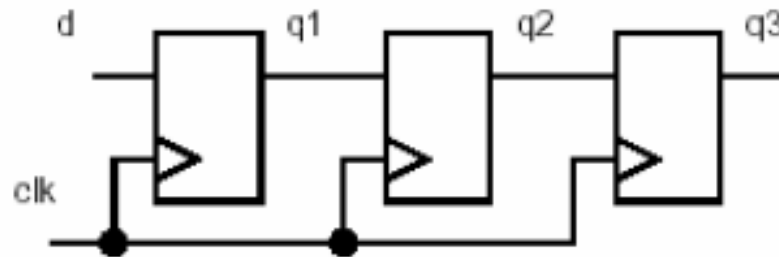


```
module fsm_mod(q1, q0, in, clk);  
  output q1, q0;  
  input clk, in;  
  reg q1, q0;  
  
  always @(posedge clk) begin  
    q1 <= in;  
    q0 <= in | q1;  
  end  
endmodule
```



# MODELING SEQUENTIAL LOGIC USING BEHAVIORAL MODELING

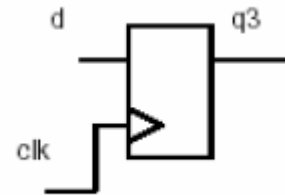
- Task: Implement the following register



# MODELING SEQUENTIAL LOGIC USING BEHAVIORAL MODELING

- Using Blocking Assignment

```
module pipeb1 (q3, d, clk);  
    output [7:0] q3;  
    input  [7:0] d;  
    input          clk;  
    reg    [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q1 = d;  
        q2 = q1;  
        q3 = q2;  
    end  
endmodule
```



- On every clock edge, the input value is transferred directly to the q3-output without delay. This clearly does not model a pipeline register and will actually synthesize to a single register!



# MODELING SEQUENTIAL LOGIC USING BEHAVIORAL MODELING

- Using Blocking Assignment

```
module pipeb2 (q3, d, clk);  
    output [7:0] q3;  
    input  [7:0] d;  
    input      clk;  
    reg  [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q3 = q2;  
        q2 = q1;  
        q1 = d;  
    end  
endmodule
```

- The blocking assignments have been carefully ordered to cause the simulation to correctly behave like a pipeline register. This model synthesizes to the required pipeline register
- It works but **BAD DESIGN**



# MODELING SEQUENTIAL LOGIC USING BEHAVIORAL MODELING

- Using Non-Blocking Assignment

```
module pipen1 (q3, d, clk);  
    output [7:0] q3;  
    input  [7:0] d;  
    input          clk;  
    reg    [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q1 <= d;  
        q2 <= q1;  
        q3 <= q2;  
    end  
endmodule
```

- Each of the two blocking-assignment examples is rewritten with non-blocking assignments, each will simulate correctly and synthesize the desired pipeline logic.



# MODELING SEQUENTIAL LOGIC USING BEHAVIORAL MODELING

- Using Non-Blocking Assignment

```
module pipen2 (q3, d, clk);  
    output [7:0] q3;  
    input  [7:0] d;  
    input          clk;  
    reg    [7:0] q3, q2, q1;  
  
    always @(posedge clk) begin  
        q3 <= q2;  
        q2 <= q1;  
        q1 <= d;  
    end  
endmodule
```

- Each of the two blocking-assignment examples is rewritten with non-blocking assignments, each will simulate correctly and synthesize the desired pipeline logic.



## GUIDELINE # 1

**When modeling  
sequential logic, use  
non-blocking ('<=')  
assignments.**





# MODELING COMBINATIONAL LOGIC USING BEHAVIORAL MODELING

```
module ao4 (y, a, b, c, d);  
    output y;  
    input  a, b, c, d;  
    reg    y, tmp1, tmp2;  
  
    always @(a or b or c or d) begin  
        tmp1 <= a & b;  
        tmp2 <= c & d;  
        y    <= tmp1 | tmp2;  
    end  
endmodule
```

- Non-blocking assignments evaluate the RHS expressions before updating the LHS variables. The outputs will reflect the old values of tmp1 and tmp2, not the values calculated in the current pass of the always block.



## GUIDELINE # 2

**When modeling  
combinational logic, use  
blocking ('=')  
assignments.**



# MIXED SEQUENTIAL & COMBINATIONAL LOGIC

```
module mix(q, a, b, clk, rst_n);  
    input clk, rst_n, a, b;  
    output q;  
    reg q;  
    always @ (posedge clk or negedge rst_n)  
        begin  
            if (!rst_n) q <= 1'b0;  
            else q <= a ^ b;  
        end  
endmodule
```

Use Non-blocking

- It is sometimes convenient to group simple combinational equations with sequential logic equations.



## GUIDELINE # 3

**When modeling **both**  
sequential and combinational  
logic within the same  
always block, use  
non-blocking('<=')  
assignments.**



# MIXED SEQUENTIAL & COMBINATIONAL LOGIC

```
module mix(q, a, b, clk, rst_n);  
    input clk, rst_n, a, b;  
    output q;  
    reg q, y;  
    always @ (*)  
        y = a ^ b;  
    always @ (posedge clk or negedge rst_n)  
    begin  
        if (!rst_n)    q <= 1'b0;  
        else q <= y;  
    end  
endmodule
```

## Preferred Way

Use Blocking

Use Non-blocking

- The same logic that is implemented in the previous Example could also be implemented as two separate always blocks, one with purely combinational logic coded with blocking assignments and one with purely sequential logic coded with non-blocking assignments.



## GUIDELINE # 4

**Do not mix  
blocking and non-blocking  
assignments in the same  
always block.**



# MULTIPLE ASSIGNMENTS

```
module badcode1 (q, d1, d2, clk, rst_n);  
    output q;  
    input  d1, d2, clk, rst_n;  
    reg    q;  
  
    always @(posedge clk or negedge rst_n)  
        if (!rst_n) q <= 1'b0;  
        else        q <= d1;  
  
    always @(posedge clk or negedge rst_n)  
        if (!rst_n) q <= 1'b0;  
        else        q <= d2;  
endmodule
```

We have problem here



## GUIDELINE # 5

**Do not make assignments  
to the same variable  
from more than one  
always block.**





# SYNCHRONOUS AND ASYNCHRONOUS

## ○ Synchronous/asynchronous reset/set

```
module dff (clk, set, reset, d, q);  
    input  clk, set, reset, d;  
    output q;  
    reg    q;
```

```
    always @(posedge clk)  
        if (reset)    q = 1'b0;  
        else if (set) q = 1'b1;  
        else          q = d;
```

```
endmodule
```

```
module dff (clk, set, reset, d, q);  
    input  clk, set, reset, d;  
    output q;  
    reg    q;
```

```
    always @(posedge clk or posedge reset)  
        if (reset)    q = 1'b0;  
        else if (set) q = 1'b1;  
        else          q = d;
```

```
endmodule
```



# ASYNCHRONOUS RESET

always @ (posedge clock or negedge reset)

- So whenever reset is asserted, the registers reset without waiting for positive edge of clock



## case STATEMENT

- Requires complete bitwise match over all four values so expression and case item expression must have same bit length
- Example:

```
always @ (state)
begin
    case (state)
        2'b00: next_state = s1;
        2'b01: next_state = s2;
        2'b10: if (x) next_state = s0;
               else next_state = s1;
        default: next_state = 1'bxx;
    endcase
end
```



# VARIANTS OF case STATEMENT

- `casex` and `casez`
- `casez` – z is considered as a don't care
- `casex` – both x and z are considered as don't cares

- Example:

```
casez (X)
```

```
  2'b1z: A = B + C;
```

```
  2'b11: A = B / C;
```

```
endcase
```



## casex STATEMENT

- Requires bitwise match over all bits except for positions containing x; executes first match encountered if multiple matches.

- Example:

```
always @ (code)
```

```
begin
```

```
    casex (code)
```

```
        2'b0x: control <= 8'b00100110; //00 and 01
```

```
        2'b10: control <= 8'b11000010;
```

```
        2'b11: control <= 8'b00111101;
```

```
        default: control <= 8b'00000000;
```

```
    endcase
```

```
end
```



## casez STATEMENT

- Requires bitwise match over all except for positions containing z or ? (? is explicit don't care); executes first match encountered if multiple matches.

- Example:

```
always @ (code)
```

```
begin
```

```
    casez (code)
```

```
        2'b0z: control <= 8'b00100110; //00 and 01
```

```
        2'b10: control <= 8'b11000010;
```

```
        2'b11: control <= 8'b00111101;
```

```
        default: control <= 8b'00000000;
```

```
    endcase
```

```
end
```



## if ... else STATEMENT

- Must be careful to define outcome for all possible conditions – failure to do so can cause unintentional inference of latches!
- `else` is paired with nearest `if` when ambiguous - use `begin` and `end` in nesting to clarify.
- Nested `if ... else` will generate a “serial” or priority like circuit in synthesis which may have a very long delay - better to use `case` statements to get “parallel” circuit.



# for LOOP EXAMPLE

- Example:

```
integer r, i;
```

```
initial
```

```
begin
```

```
    r = 0;
```

```
    for (i = 1; i <= 7; i = i + 2)
```

```
        begin
```

```
            r[i] = 1;
```

```
        end
```

```
    end
```





# while LOOP EXAMPLE

initial

begin

r = 0;

i = 0;

while (i <= 7)

begin

r[i] = 1;

i = i + 2;

end

end



# forever LOOP EXAMPLE

```
initial
begin
    clk = 0;
    forever
        begin
            #50 clk = 1;
            #50 clk = 0;
        end
    end
end
```

- Usually used in test-benches rather than for synthesized logic.
- Runs until the simulation stops



# PROCEDURAL TIMING, CONTROLS & SYNCHRONIZATION (FIO)

## ○ **wait** Construct

- Suspends activity in behavior until expression following **wait** is TRUE

## ○ Example:

always

begin

a = b;

c = d;

**wait** (advance); //will go to next line when advance ==1

end

