

---

# Introduction to VeriLog

## Lab Session # 2



UNIVERSITY of  
ROCHESTER

---

ECE 200: Computer Organization

Raj Parihar

[parihar@ece.rochester.edu](mailto:parihar@ece.rochester.edu)

# Outline

- Background
- Introduction of HDL
- VeriLog
- HDL Design Flow
- VeriLog Tutorial
- Key Concepts
- Things to Learn
- Design Example

# Background

- What you should know?
  - Basic of Digital Logic Design
  - Programming in C or in any HLL
  - Create/ Compile Projects in ModelSim
- Don't worry, if you are not an adept in above things
  - We will learn more as we proceed along
- At the end of the Semester
  - You would be able to design a general purpose processor

# What is HDL?

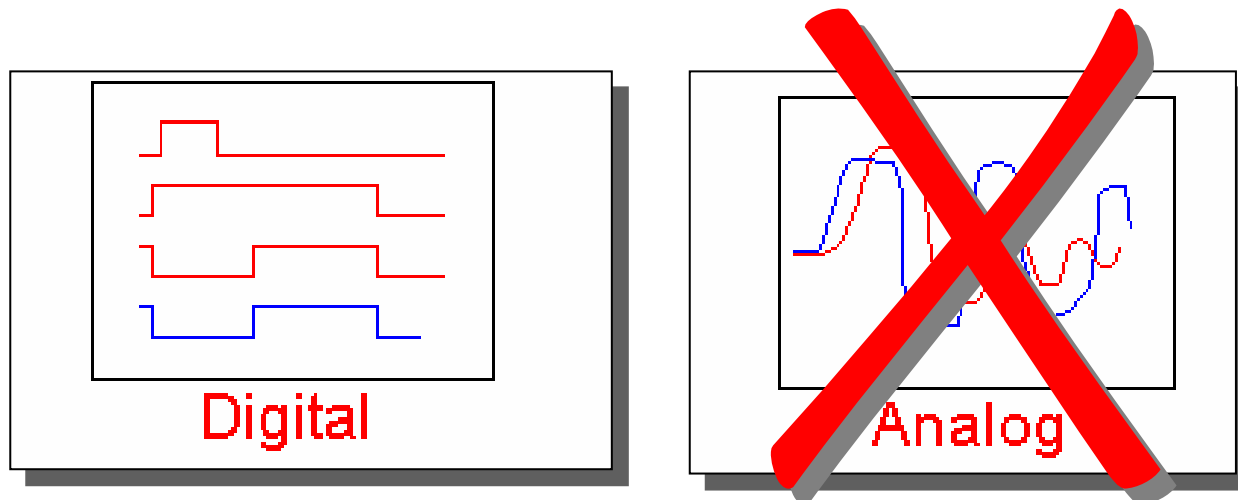
- “**H**ard & **D**ifficult **L**anguage”
  - No, it is “**H**ardware **D**escription **L**anguage”
- High Level Language
  - To describe the circuits by syntax and sentences
  - As oppose to circuit described by schematics
- Two widely used HDLs
  - VeriLog – Similar to C
  - VHDL – Similar to PASCAL

# VeriLog

- One of the TWO widely used HDL
  - We would use in these labs
- Developed in 1984
  - In 1995: IEEE std: 1364
  - Easier than VHDL and other HDLs

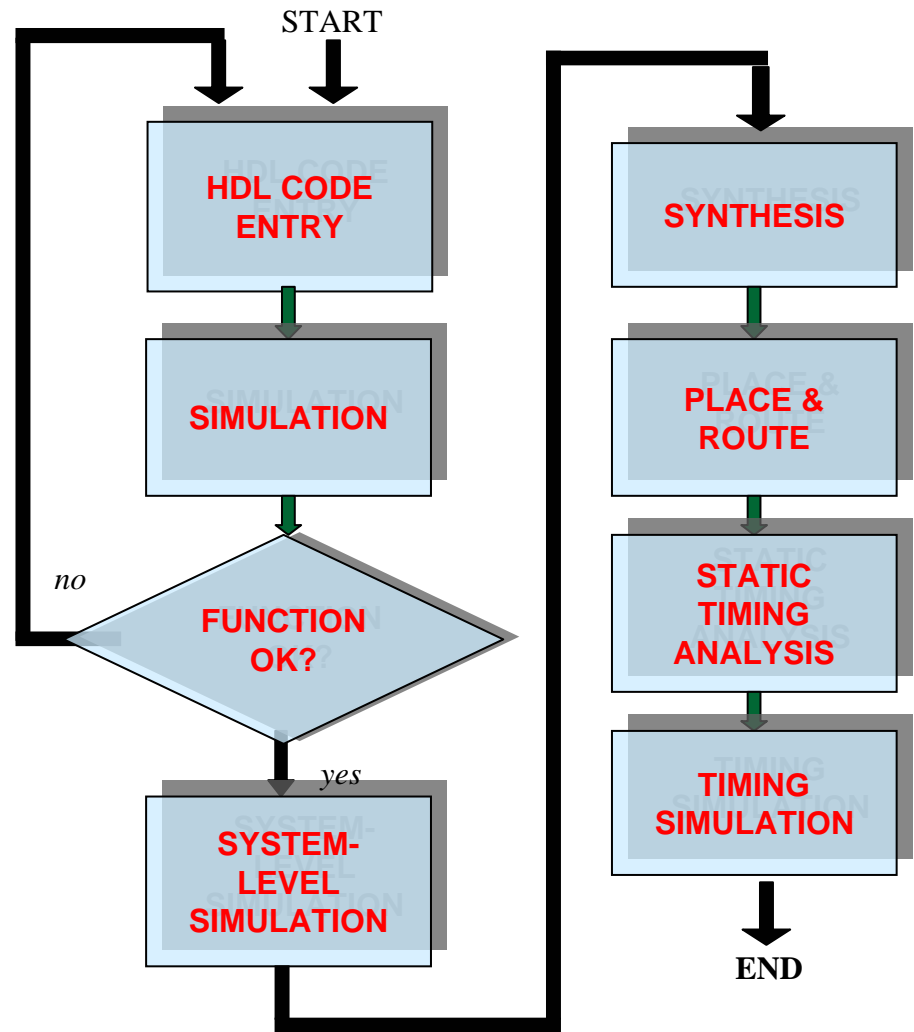
# Basic Limitation of Verilog

Description of digital systems only



# HDL Design Flow

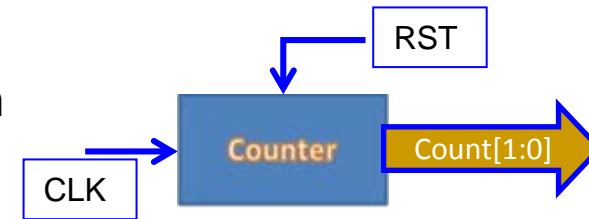
- Enter your design
  - As HDL Code
- Run Simulation
  - If OK, Proceed
  - Else, Change HDL
- Synthesis
  - Equivalent Hardware
- Timing Simulation
  - If Pass, finish
  - Else, Change HDL



# Types of Modeling

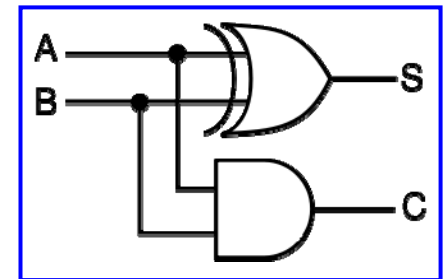
## ■ Behavioral Modeling

- ❑ Describes the functionality of a component/system
- ❑ Use of **if, else** kind of statements
- ❑ Example: in Lab # 1: 2-bit binary counter



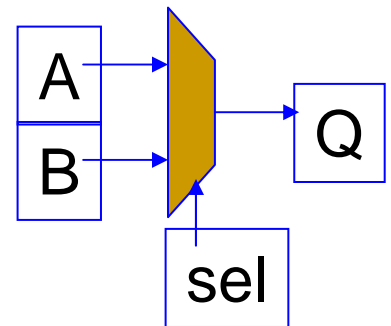
## ■ Structural Modeling

- ❑ A component is described by the interconnection of lower level components/primitives
- ❑ Use lower level primitives i.e. **AND, OR, XOR**
- ❑ Example: in Lab # 1: Half adder design



## ■ Data Flow Modeling

- ❑ Explicit relation between OUTPUT and INPUT
- ❑ Use of Boolean or **assign** expression
- ❑ Example: in Lab # 1: 2:1 MUX design





# VeriLog Tutorial

Fasten your seatbelt!

# VeriLog Basics

- Case sensitive language
  - temp, Temp, TEMP are THREE different names
- Keywords are lower case
  - module, endmodule, input, output, reg etc.
- Comments are specified as
  - // This is a single line comment
  - /\* This is how a multiple-line  
comment is specified in VeriLog..... \*/
  - Multiple line comments can't be nested
    - i.e. /\* Outer /\*Inner Comment\*/ Comment \*/
- Semicolons(;) are line terminators
- Notice the similarity to 'C'

# Gate Level: Structural Modeling

- Specify the structure of design
  - Gate primitives and Connecting wires

//Example of 2:1 mux

**module** mux (y, a, b, sel );

**output** y ;

**input** a, b, sel ;

**wire** y1, y2, sel\_n ;

**not** A1 (sel\_n, sel);

**and** A2 (y1, a, sel\_n);

**and** A3 (y2,b, sel);

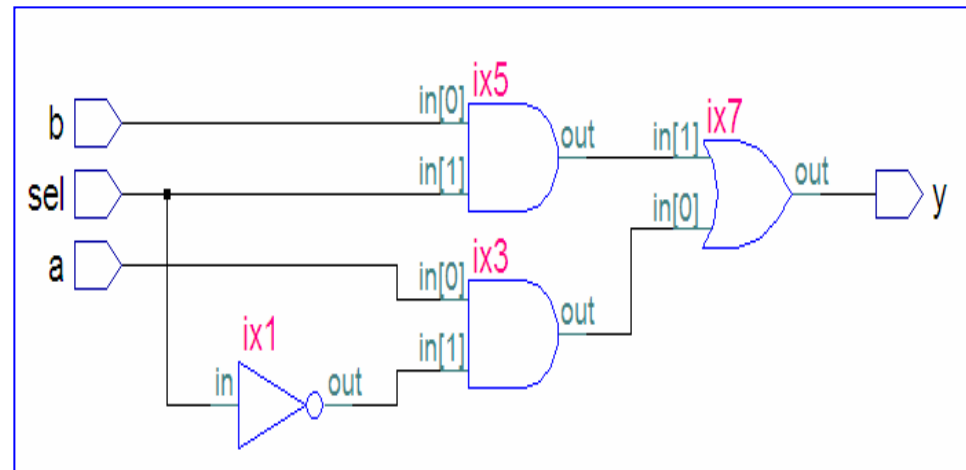
**or** A4 (y,y1,y2);

**endmodule**

Single Line Comment

Component Interface

Gate &  
wires

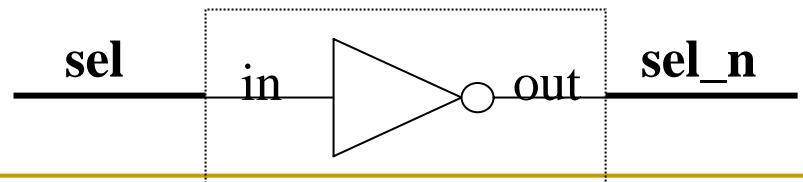


# Instance

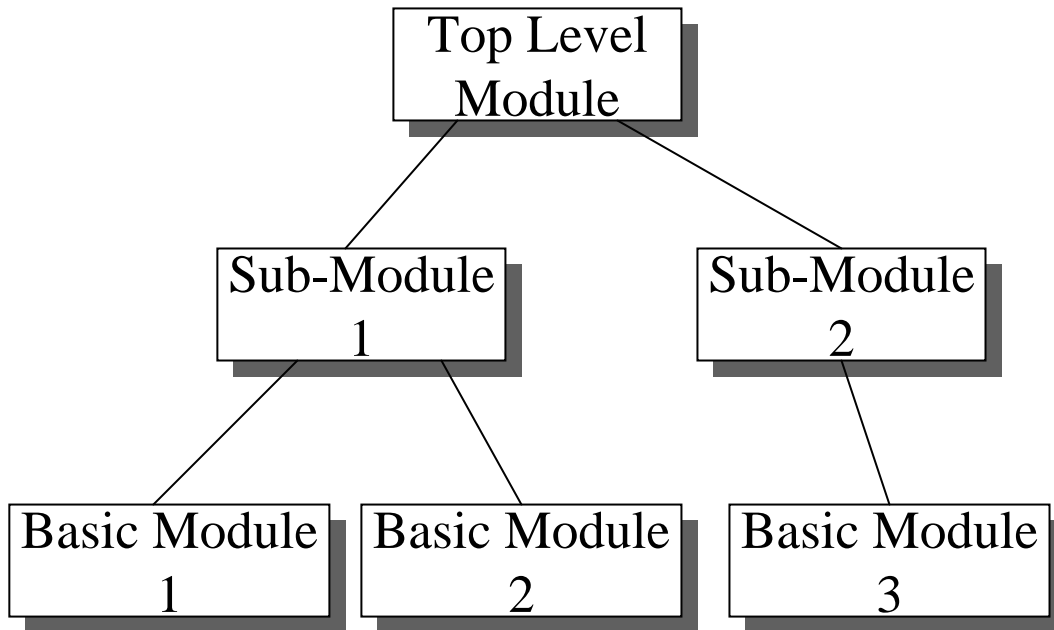
- Instantiation is the process of calling a module
  - 4 gates are instantiated in our MUX example
- The referred module are called instances
  - 2 instance of AND & 1 instance each of OR & NOT

```
module mux (y, a, b, sel );  
    output y ;  
    input  a, b, sel ;  
    wire y1, y2, sel_n ;  
    not A1 (sel_n, sel);  
    and A2 (y1, a, sel_n);  
    and A3 (y2, b, sel);  
    or  A4 (y, y1, y2);  
endmodule
```

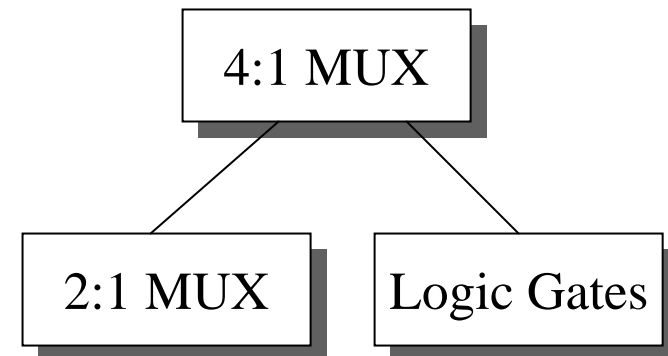
- **Instance name**
  - A1, A2, A3 & A4
- **Positional Instantiation**
  - not (out, in)
  - not (sel\_n, sel)



# Hierarchical Design



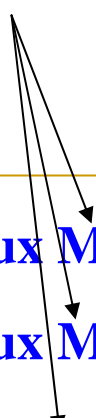
E.g.



# Hierarchal Design: Named Instances

```
module mux4_1 (y, a, b, c, d, sel0, sel1 );  
    output y ;  
    input  a, b, c, d, sel ;  
    wire y1, y2 ;  
  
    mux M1 (.y(y1), .a(a), .b(b), .sel(sel0));  
    mux M2 (.y(y2), .a(c), .b(d), .sel(sel0));  
    mux M3 (.y(y), .a(y1), .b(y2), .sel(sel1));  
  
endmodule
```

Named instances  
are recommended



```
mux M1 (y1, a, b, sel0);  
mux M2 (y2, c, d, sel0);  
mux M3 (y, y1, y2, sel1);
```

# Vectors: Group of Signals

- Also known as BUS in hardware

```
module mux4_1 (y, in, sel );
```

Syntax [MSB:LSB]

```
    output y ;
```

```
    input  [3:0] in ;
```

```
    input  [1:0] sel
```

```
    wire y1, y2 ;
```

```
    wire [1:0] sel;
```

2-bit wide bus



Bit Select of a bus



```
    wire [3:0] in
```

```
    mux M1 (.y(y1), .a(in[0]), .b(in[1]), .sel(sel[0]) );
```

```
    mux M2 (.y(y2), .a(in[2]), .b(in[3]), .sel(sel[0]) );
```

```
    mux M3 (.y(y), .a(y1), .b(y2), .sel(sel[1]) );
```

```
endmodule
```

# Register Transfer Level (RTL)

- A way of describing the operation of synchronous digital circuit
  - Behavior is described in terms of state of registers
  - Widely used in complex system design
- In RTL design, a circuit's behavior is defined
  - in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals



# Number System

- Syntax : <size>'<base\_format>number
- size : specifies number of bits in DECIMAL
  - optional. Default number atleast 32
- base\_format : specifies the radix
  - optional. Default is decimal
  - d or D for decimal
  - h or H for hexadecimal
  - b or B for binary
  - o or O for octal
- Underscore (\_) is allowed to improve readability
  - cannot be the first character

# Number System: Example

## ■ Example for sized number : 30

- ❑ binary : 6'b01\_1110 same as 6'b11110
- ❑ octal : 6'o36
- ❑ decimal : 6'd30
- ❑ hexadecimal : 6'h1E

## ■ Example of un-sized number

- ❑ No size format : 30 (decimal number 30, 32-bit wide)
- ❑ No size : 'h1E (in hexadecimal, 32-bit wide)

# Value Set

- $0$  represents low logic level or false condition
- $1$  represents high logic level or true condition
- $x$  represents unknown logic level
- $z$  represents high impedance logic level

# Continuous Assignment

- Syntax : **assign** LHS = RHS ;
  - wire out ;
  - assign out = a | b ;
- LHS must be a **net** declared before usage
- RHS can be **reg**, **net** or **function**
- LHS is evaluated whenever RHS changes
- Implicit assignment
  - short cut. Combines declaration and assignment
    - wire out = a | b ;

# Arithmetic Operators

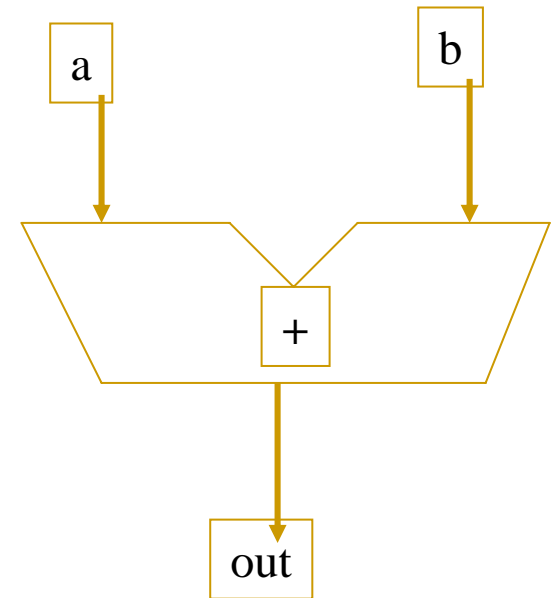
- Add (+)
- Subtract (-)
- Multiply (\*)
- Divide (/)
- Modulus (%)
  - remainder from division
  - takes sign of first operand
  - Examples
    - $13 \% 3$  equals 1
    - $9 \% -2$  equals 1
    - $-9 \% 2$  equals -1

# Realization of Arithmetic Operator

- Modulus and division operator
  - supported only if operands are constant
- Addition, subtraction and multiplication
  - are fully supported by synthesis tool

- Example

```
module adder (out, a , b) ;  
input [3:0] a, b ;  
output [3:0] out ;  
    wire [3:0] out = a + b ;  
endmodule
```



# Logical Operations

- Logical Boolean Operation
  - NEGATION ( ! )
  - Logical AND ( && )
  - Logical OR ( || )
- Result is always 1 (true), 0 (false) or x (unknown)
  - Operand is zero if it's value equals zero.
  - Operand value is one if its value is other than zero.
  - Operand itself can be a result of another expression.

# Relational Operators

- Greater than ( $>$ )
- Less than ( $<$ )
- Greater than or equal ( $>=$ )
- Less than or equal ( $<=$ )
- Equality ( $==$ )
- Inequality ( $!=$ )

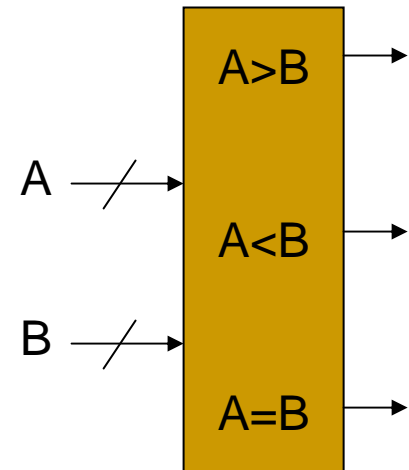
- Isn't it similar to C?



# Example: 4-bit Magnitude Comparator

## //4-bit magnitude comparator

```
module mag_comp (a_lt_b, a_gt_b, a_eq_b, a, b) ;  
output  a_lt_b, a_gt_b, a_eq_b ;  
input   [3:0] a, b ;  
wire a_lt_b, a_gt_b, a_eq_b ;  
    assign a_lt_b = ( a < b ) ;  
    assign a_gt_b = ( a > b ) ;  
    assign a_eq_b = ( a == b ) ;  
endmodule
```



# Bitwise Operations

- NOT ( $\sim$ )
- AND ( $\&$ )
  - if one input value is 'x and the other 0, result value is 0
  - if one input value is x and the other 1, result value is x
- OR ( $\|$ )
  - if one input value is x and the other 0, result value is x
  - if one input value is x and the other 1, result value is 1
- XOR ( $\wedge$ )
  - any input value is x, result value is x

# Some More Bitwise Operations

- XNOR ( $\wedge\sim$  or  $\sim\wedge$ )
  - any input value is 'X' result value is 'X'
- NAND ( $\sim\&$ )
  - if one input value is 'x' and the other 0, result value is 1
  - if one input value is x and the other 1, result value is x
  - ( $\&\sim$ ) is not allowed
- NOR ( $\sim|$ )
  - if one input value is x and the other 0, result value is x
  - if one input value is x and the other 1, result value is 0
  - ( $| \sim$ ) is not allowed

# Shift Operator

- Right shift ( $\gg$ )
- Left shift ( $\ll$ )
  - Do not wrap around
  - Vacant bit positions are filled with zero

- Examples

- `wire [3:0] out = 4'b1010 >> 1`
  - Result is 4'b0101. **Division**
- `wire [3:0] out = 4'b0010 << 1`
  - Result is 4'b0100. **Multiplication**
- `wire [3:0] out = 4'b0010 << 8`
  - Result is 4'b0000

Good alternative for  
costly “multiplication”  
and “Division” operations

# Concatenation Operator

- Concatenation Operator (`{}`)
- Allows appending of multiple operands
- Operands must be sized
  - `wire [7:0] out = {4, 6}` not allowed
- Shift Register, S2P Conversion, Barrel Shifter
- Example. `A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110`
  - `Y = { A, B, C, D, 4'b1111};`
    - Answer is `12'b1001_0110_1111`
  - `Y = { A, B[0], C[1]}`
    - Result is `3'b101`
- So far, many things are like C 😊

# Conditional Operator

- Condition Operator ( ?: )
  - ternary operator
  - condition ? True\_statement : false\_statement
  - equivalent to if else
- Conditional operators can be nested.
- Example
  - assign out = sel ? a : b ; //2-to1 multiplexer
  - assign out = en ? a : 1'bz ; //Tristate buffer
  - assign out = sel1 ? A : sel2 ? B : C ; //3-to-1 mux

# Parameters

- Declare runtime constant.
  - **parameter** p1 = 8, real constant = 2.309 ;
- Parameters are local to a module.
- Used to configure design values
- **defparam** : override parameter value
  - Specify the complete hierarchy
- New value within module instance using #.
  - Multiple parameter order same as declared in module
  - Not necessary to assign new values to all parameters
  - cannot skip over a parameter

# Structured Procedure

- Provide means to specify concurrency
  - Real hardware operates concurrently
  - Differentiates HDL from other programming language
    - Programming Languages are sequential
- **always** and **initial**
  - initial cannot be synthesized
- **Concurrency** is not explicit in C



# Always Block

- always block run **concurrently**.
- Can have one or multiple procedural assignment statements.
  - Multiple statements must be enclosed within **begin end**
- Procedural assignment statement run **sequentially**
- sensitivity list decides when always block is entered
- Syntax :

```
always @(sensitivity_list)
begin
    //multiple/single procedural assignment
    //statement
end
```

# Procedural Assignments

- Blocking
  - $LHS = RHS ;$
- Non-Blocking
  - $LHS \leq RHS ;$
- LHS has to be of type reg
  - Common compilation error is to declare LHS as net.
  - LHS value is held until driven again
- RHS rules : Same as in continuous assignment
  - can be an expression using all the previous operators.

# Blocking Assignments

- **LHS = RHS ;**
- Evaluate (RHS), assign (value to LHS), proceed
  - Equivalent to variable assignment in VHDL :=
- Execution order as specified in block statement
- BLOCKS only in that concurrent block
- Does not BLOCK in other concurrent block
- SHOULD be used in models and testbench
  - Simulation efficient

# Non-Blocking Assignments

- **LHS <= RHS ;**
- Evaluate (RHS), schedule (value to LHS), proceed
  - values are assigned at the end of current simulation time
  - Equivalent to signal assignment in VHDL
- Execution order is independent of statement order
- Recommended for RTL
  - Models hardware behavior
- Avoids race-condition
  - same identifier assigned different values in different process.
- For synthesis mix is **not allowed** in **same** always.

# Example: Blocking and Non-Blocking

## Assignments

A = 3 ;  
B = 4 ;  
A <= 3 + 4 ;  
C = A ;

A = 3 ;  
B = 4 ;  
A = 3 + 4 ;  
C = A ;

Non-Blocking → Parallel  
Blocking → Sequential

A=2;  
A <= 3 ;  
B <= 4 ;  
A <= 3 + 4 ;  
C <= A ;

## Results

A = 7 ;  
B = 4 ;  
C = 3 ;

A = 7 ;  
B = 4 ;  
C = 7 ;

A = 7 ;  
B = 4 ;  
C = 2 ;

---

# Break: 10 Minutes

---

Drink Water and Stretch your Legs!

# Conditional Statements

- **if** (<condition>) true\_statement(s)
- **if** (<condition>)  
    true\_statement(s)  
    **else**  
        false\_statement(s)
- **if** (<condition>)  
    true\_statement(s)  
    **else if** (<condition\_1>)  
        true\_statement(s)  
    **else**  
        false\_statement(s)

Group multiple statements  
using **begin** and **end**

**In VeriLog:**

**Begin & end → { }**

Whereas

**{ } is used for**

.....

# Case Statements

- Multi-way Branching

- Syntax : **case** (expression)

alternative\_1 : statement(s)\_1;

alternative\_1 : statement(s)\_1;

.....

**default** : default\_statement(s);

**endcase**

- Default statement (s) is executed when none of the branch condition are met.



# Two Different Implementation

//2-to1 mux

```
module mux (y, sel, a, b);  
input sel, a, b ;  
output y ;  
reg y ;  
always @(sel or a or b)  
begin  
    if (sel)  
        y <= a ;  
    else  
        y <= b ;  
end  
endmodule
```

//2-to1 mux

```
module mux (y, sel, a, b);  
input sel, a, b ;  
output y ;  
reg y;  
always @(sel or a or b)  
begin  
    case (sel)  
        1'b0 : y = a ;  
        1'b1: y = b ;  
        default : y =a ;  
    endcase  
end  
endmodule
```

Which one is  
Blocking and  
which one is  
Non-Blocking?

# Behavioral Modeling

- Used for big systems
  - Structural requires too many components
  - Closest to traditional 'C'
- Mainly used for Memory Sub-systems
- Not all behavioral codes are **Synthesizable**
  - i.e. wait statement

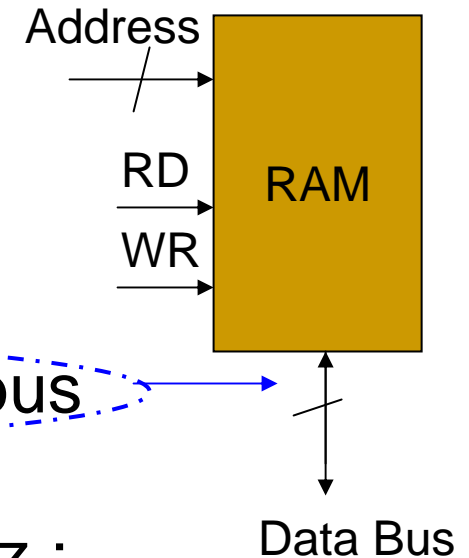
# Arrays

- 1 dimensional and 2 dimensional
  - **reg** index [0:7] ; // 1 dimensional array;
  - 1D not same as vector/bus wire [7:0] index ;
  - bus -> row matrix, 1D -> column matrix
- Array cannot be of type real
- 2-D array used to model memory (RAM/ROM)
  - **reg** [MSB:LSB] mem\_name [first\_addr : last\_addr] ;
- Memory is addressed giving the location
  - out = mem\_name[32] ;
- Cannot access a bit of given location
  - copy to temp variable

# RAM Model

## //Asynchronous RAM

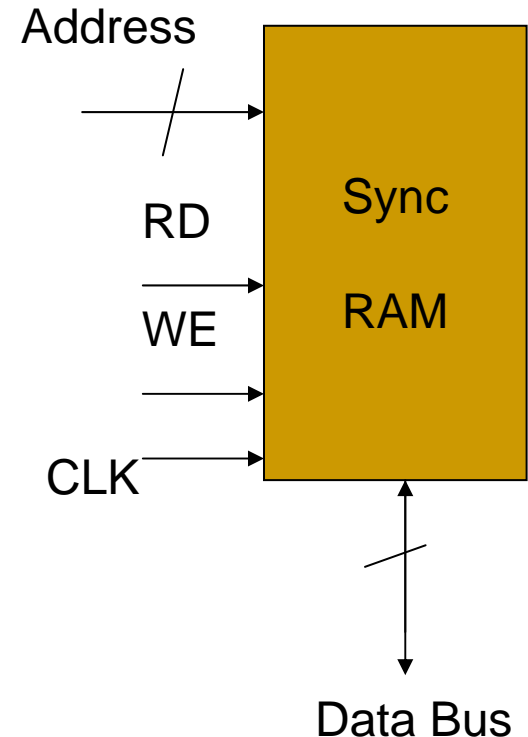
```
module ram ( dbus, addr, wr, rd ) ;  
input [4:0] addr ; //RAM depth is 32  
input wr, rd ;  
inout [7:0] dbus ; //bi-directional data bus  
reg [7:0] mem [4:0] ;  
wire [7:0] dbus = rd ? mem[addr] : 8'bz ;  
always @ ( wr) begin  
    if (wr)  
        mem[addr] = dbus ;  
endmodule
```



# Another RAM Model

## //Synchronous RAM

```
module ram ( dbus, addr, we, rd, clk );  
input [4:0] addr ; //RAM depth is 32  
input we, rd, clk ;  
inout [7:0] dbus ; //bi-directional data bus  
reg [7:0] mem [4:0] ;  
wire [7:0] dbus = rd ? mem[addr] : 8'bz ;  
always @ (posedge clk) begin  
    if (we)  
        mem[addr] = dbus ;  
end  
endmodule
```



# Timing Control

- Simple delay # delay LHS = RHS ;
- Event triggered @(signal) LHS = RHS;
- Edge triggered @(pos/negedge signal) LHS=RHS;
- Level sensitive wait (signal) LHS = RHS;
- Intra-assignment delay LHS = # delay RHS
  - delay assignment not evaluation.
  - Affects blocking statement block.

# Delays

- Symbol # specifies delay
- Delay between statement occurrence and execution
- Delay specified by numbers or identifier

**parameter** latency = 20 ;

initial

x = 0                      assigns value at time 0

# 10 y = 1                assigns at time 10

# 10 z = 1                assigns at time 20

# latency a = 0    assigns at time 20 + latency

# Delay

- Intra-assignment delay
- Delay on RHS
  - $y = \# 5 \text{ 1'b1}$
- Statement is computed at current time
- Value is assigned at specified delay



# Example: Delay and Non-Blocking

## ■ Assignment v/s intra-assignment delay

### □ Non-blocking statements

a <= 1;	a <= 1 ;
#10 b <= 0;	b <= #10 0 ;
c <= 1;	c <= 1 ;

time

<b>0</b>	a → 1
<b>10</b>	b → 0
<b>10</b>	c → 1

time

<b>0</b>	a → 1
<b>0</b>	c → 1
<b>10</b>	b → 0

# Example: Delay and Blocking

- Assignment v/s intra-assignment delay
  - blocking statements

```
a = 1;  
#10 b = 0;  
c = 1;
```

time

```
0    a → 1  
10   b → 0  
10   c → 1
```

```
a = 1    ;  
b = #10 0 ;  
c = 1    ;
```

time

```
0    a → 1  
10   c → 1  
10   b → 0
```

# Clock Generation

```
reg clock;  
  initial  
    begin  
      clock = 1'b0 ;  
      forever  
        # 10 clock = ~ clock  
    end
```

```
reg clock;  
  initial  
    begin  
      clock = 1'b0;  
      repeat(50)  
        clock = ~clock;  
    end
```

# Test Bench

- An alternative to waveform viewer
  - Similar to printf statement in 'C'
  - Useful for complex circuits

# Example: Test Bench for 2:1 MUX

```
module tbench;
reg a, b, sel ;
wire y ;
initial
begin
    a = 1'b1 ; b = 1'b0 ; sel = 1'b0 ;
    #10 sel = 1'b1 ;
    #10 a = 1'b0 ; b = 1'b1 ; sel = 1'b0 ;
    #10 sel = 1'b1 ;
end
initial
    $monitor($time, " a=%b b=%b sel=%b -> y=%b", a,b,sel,y);
mux mux_1 (y, a, b, sel ) ;
endmodule
```

## Output

```
0  a=1 b=0 sel=0 -> y=1
10 a=1 b=0 sel=1 -> y=0
20 a=0 b=1 sel=0 -> y=0
30 a=0 b=1 sel=1 -> y=1
```

---

# Key Things to Remember

- RTL coding and Behavioral Modeling
- Blocking and Non-Blocking Assignments
- Always blocks and Concurrency

# Things to Learn

- How to use ModelSim
  - ❑ How to open?
  - ❑ How to create a project?
  - ❑ Where to write the code?
  - ❑ How to execute the code?
  - ❑ How to use Wave form viewer?
  - ❑ How to write test benches?

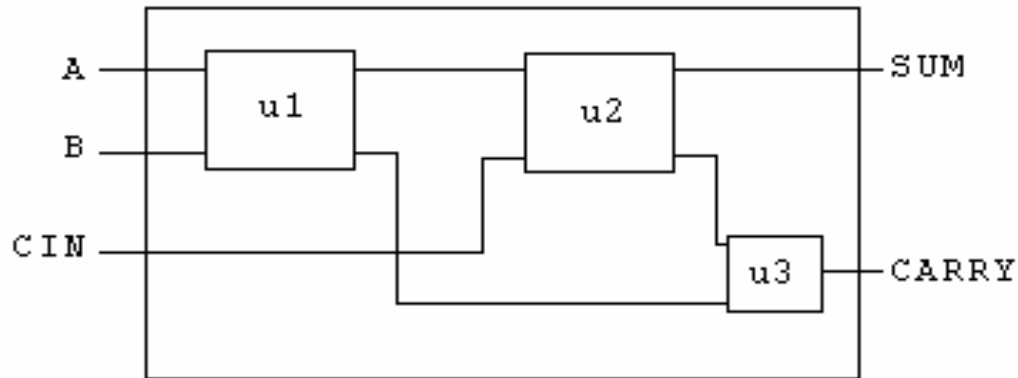
# VeriLog

---

## Summary & Key Concepts

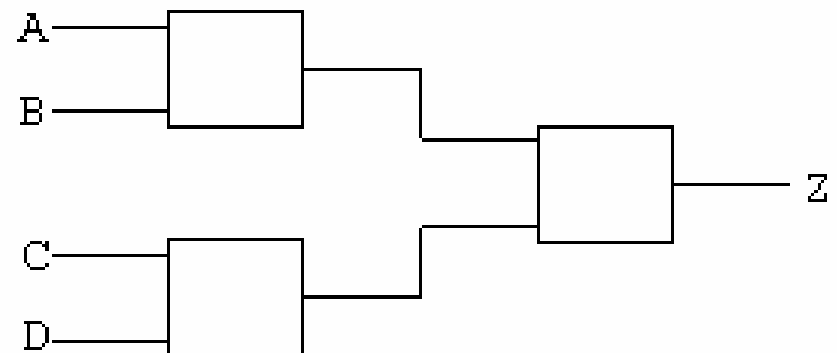


# Main Language Concepts



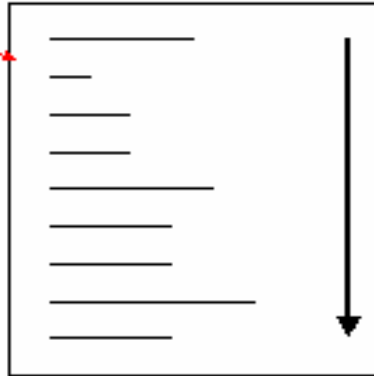
**Structures**

**Concurrency**



# Main Language Concepts

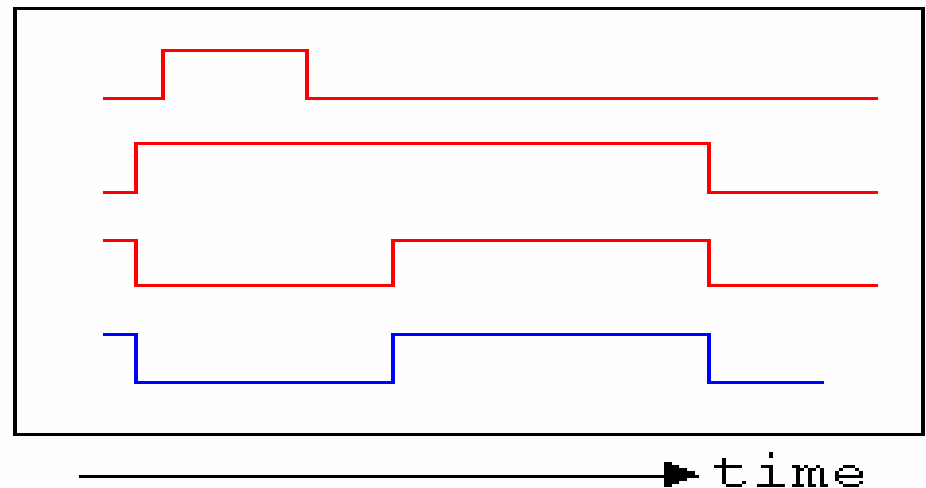
section  
of code



execution  
flow

## Procedural Statements

## Time and Delay



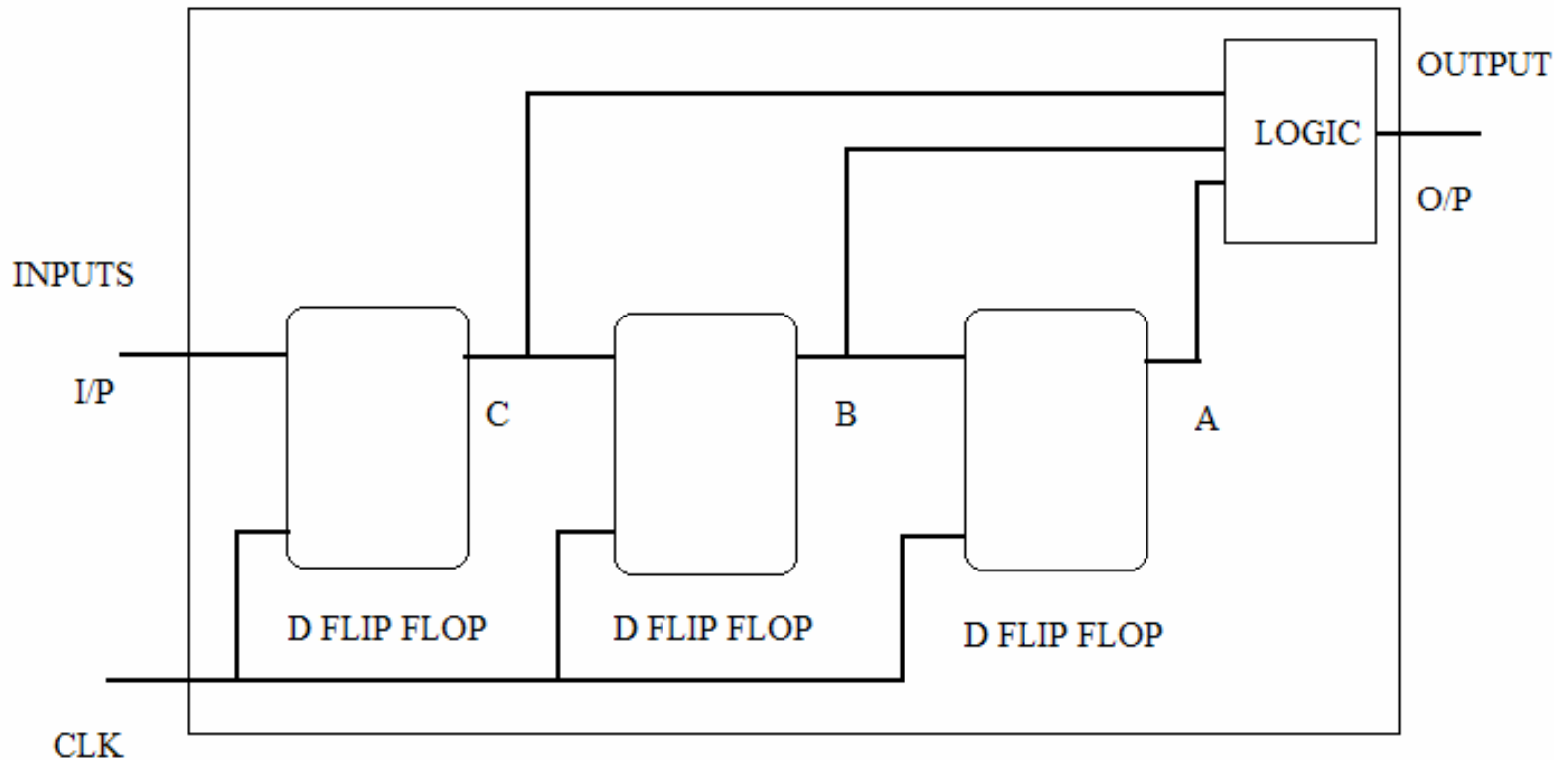
# Appendix

## Steps to Design Complete System

# Problem Statement

- Bit Pattern Detector
- Create a detector that takes input serially and produces a single bit output. Out of the last three inputs to the block, if the number of ones is greater than the number of zeros, the block must output one, else it must output zero.
- Example
  - Input : 0 0 0 1 1 0 1 1 0 1 0 0 0 0
  - Output: 0 0 1 1 1 1 1 1 0 0 0

# Detector: Block Diagram



# Bottom Up Approach

- First make a D Latch
  - Use behavioral Modeling
- Using the D latch, create D Flip Flop
  - This is hierarchical (structural) approach
- Using D Flip Flop and logic gates, Create the top level Detector
  - Again hierarchical (structural) approach

# What Next?

- Don't believe until you see it working
- Translate the design in HDL
- Create Project
- Compile and Simulate
- Verify the functions
  - Wave Form Viewer
  - Test Bench
- Welcome to the world of Digital VLSI Design

# Acknowledgement

- Slides are adopted from Prof. Anu Gupta's (Assistant Professor, BITS – Pilani, INDIA ) Presentation on “VeriLog”