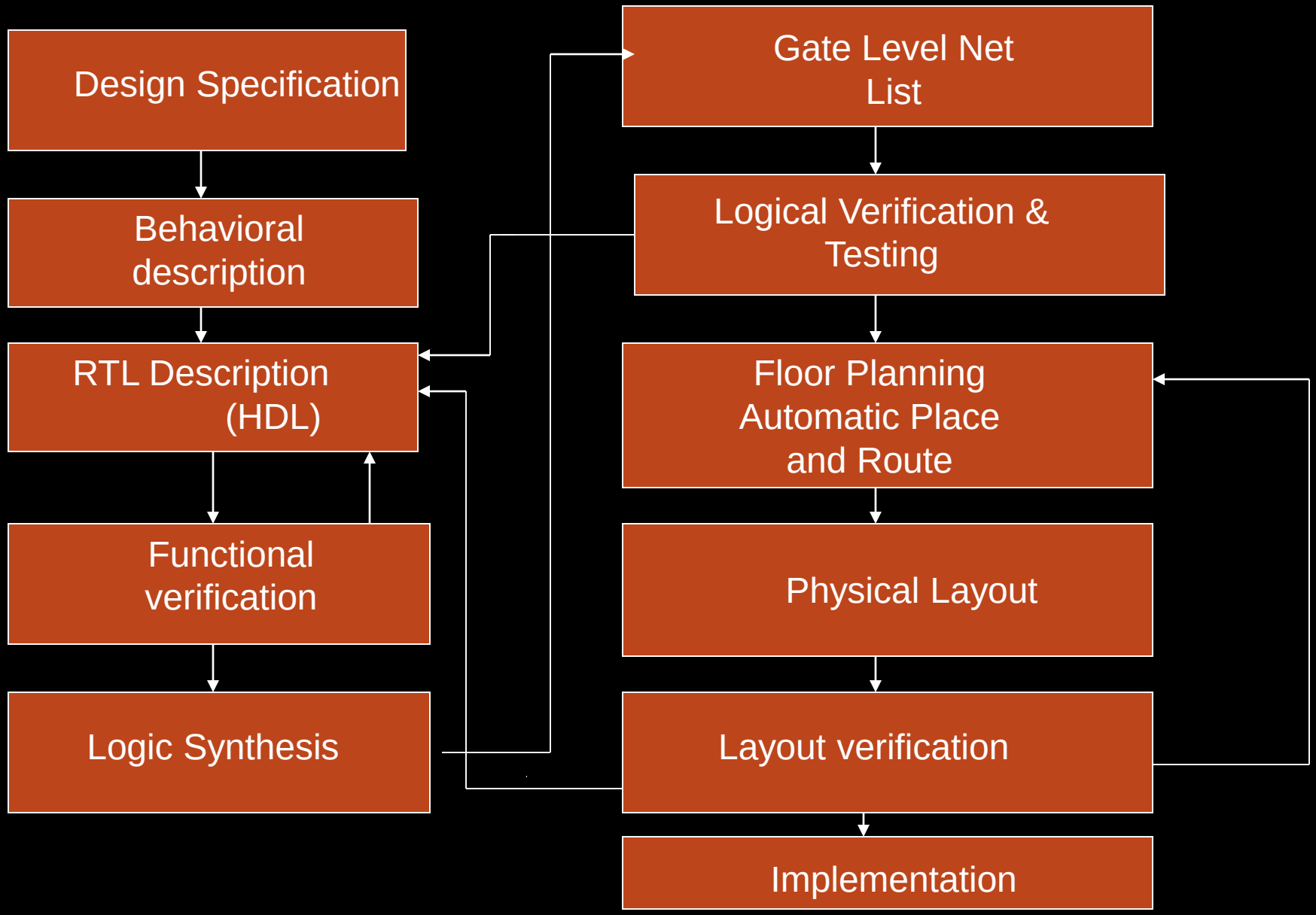
# Verilog Hardware Description Language

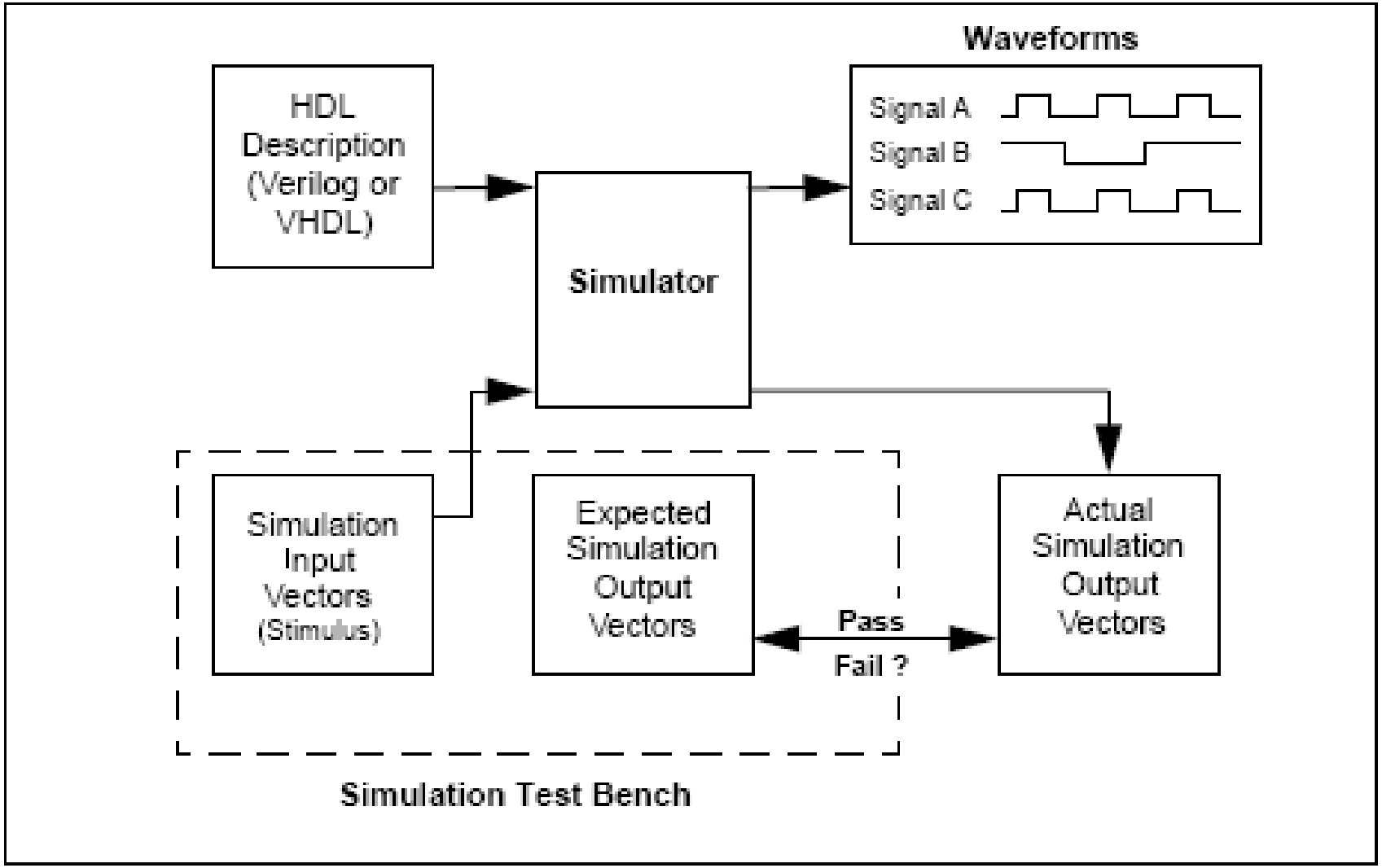
# Typical design Flow



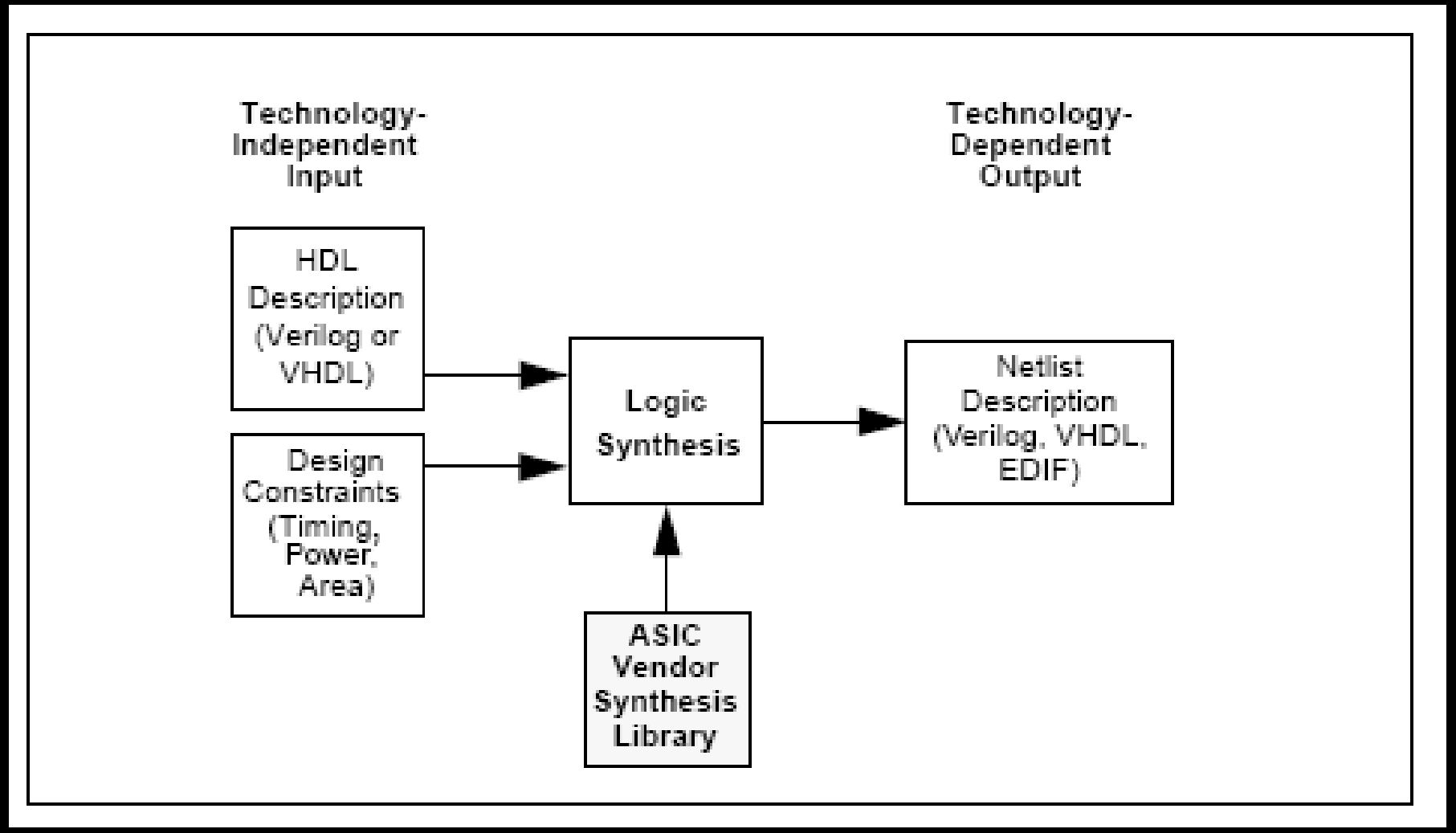
# Why HDL's ?

- Hardware Description Languages are used to model the Digital components.
- HDL Models may be represented in
  - Structural level
  - Data flow level
  - Behavioral Level
- HDL models are used for
  - Synthesis
  - Simulation
  - Documentation

# Traditional simulation process



# Logic Synthesis Process



# The Verilog Language

- Originally a modeling language for a very efficient event-driven digital logic simulator
- Later pushed into use as a specification language for logic synthesis
- Now, one of the two most commonly-used languages in digital hardware design (VHDL is the other)
- Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages
- Combines structural and behavioral modeling styles

# Basic Building Block in Verilog

module module_name (port list);

Port declarations

Parameter declarations (Interface)

‘includedirectives (Add on)

Variable declarations

Assignments

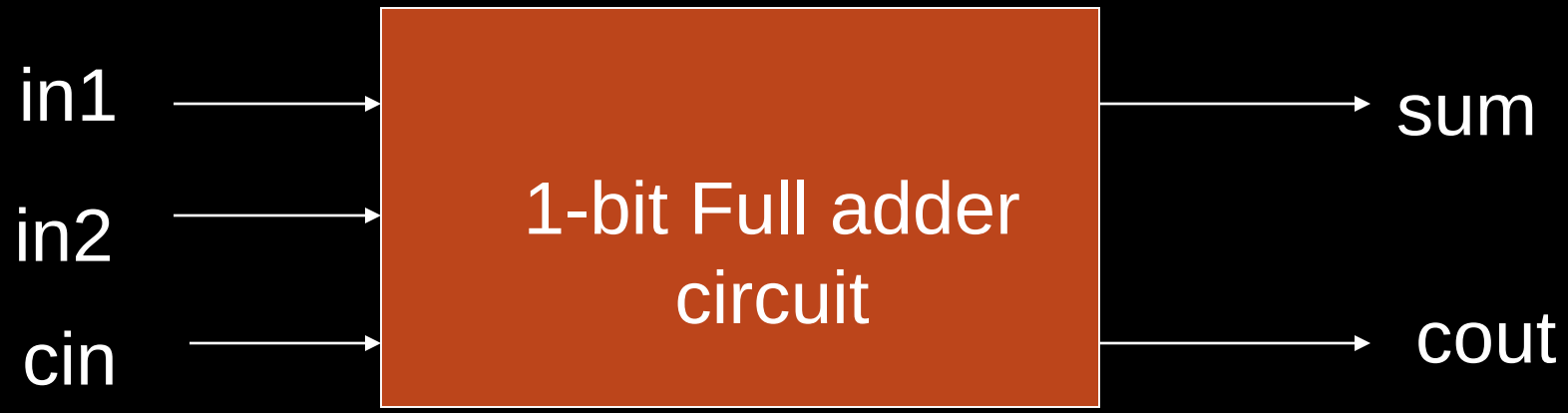
Lower level module instantiation

Initial and always block

Tasks and functions (Body)

endmodule

# Example 1-bit full adder circuit




Expression

sum = (in1 xor in2 xor cin)

cout = (in1 and in2) or (in2 and cin) or (in1 and cin)

# Structural model of a full adder circuit

```
module fadd(in1, in2, cin, sum,cout);
  input in1,in2, cin;
  output sum, cout;
  wire x1,a1,a2,a3,o1,o2;

  xor(x1,in1,in2);
  xor(sum,x1,cin);
  and(a1,in1,in2);
  and(a2,in1,cin);
  and(a3,in2,cin); or(o1, a1,a2);
  or(cout,o1,a3);

endmodule
```

# Data Flow model of a full adder circuit

```
module fadd(in1, in2, cin, sum,cout);

    input in1,in2, cin;
    output sum, cout;
    assign {cout,sum} = in1 + in2 + cin ;

endmodule
```

# Behavioral model of a full adder circuit

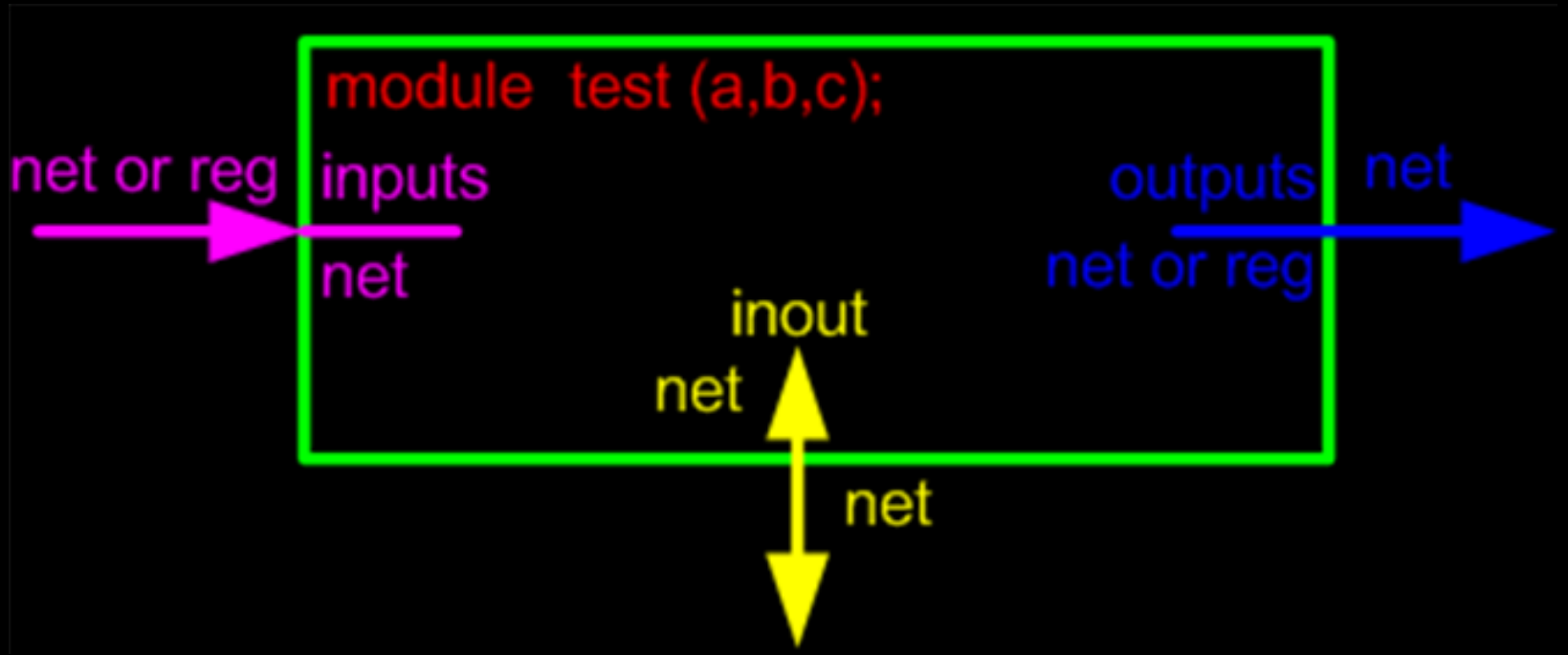
```
module fadd (in1, in2, cin, sum,cout);
 input in1,in2, cin;
 output sum, cout;
 reg sum,cout;
 always @(in1 or in2 or cin) begin
     sum = in1 ^ in2 ^ cin;
     cout = (in1 & in2 ) | (in1 & cin) | (in2 & cin);
 end
 endmodule
```

# Testbench Design

- How do we verify designs that are too large for exhaustive simulation?
  - Identify key features
    - How do we exercise?
    - What is correct response?
  - Identify "Corner cases"
    - Initial conditions
    - "Boundaries" between modes of operation
  - Ensure code coverage
    - Does testbench exercise all of the Verilog code?
    - Commerical coverage tools help this process
  - Use random inputs to test unanticipated cases

# Port Connection Rules

- Inputs : Internally must always be of type net, externally the inputs can be connected to a variable of type reg or net.
- Outputs : Internally can be of type net or reg, externally the outputs must be connected to a variable of type net.
- Inouts : Internally or externally must always be type net, can only be connected to a variable net type.



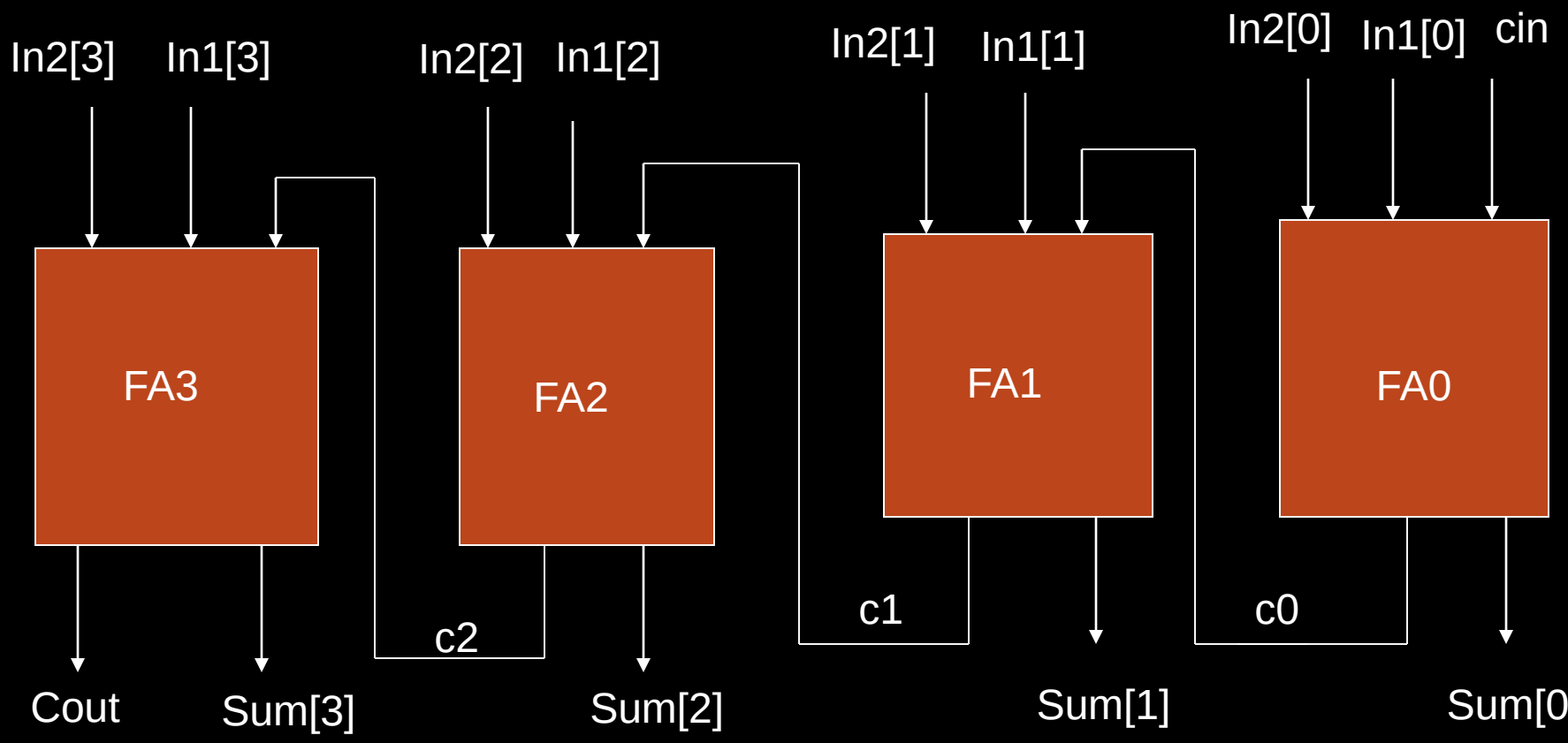
# Testbench for 1 bit Full Adder

```
module test_fadd ();
    reg in1,in2, cin;
    wire sum, cout;

    fadd dut_beh(.in1(in1), .in2(in2), .cin(cin), .sum(sum), .cout(cout));

    initial begin
      #1 in1 = 0; in2 = 0; cin = 0;
      #1  in1 = 0; in2 = 0; cin = 1;
      #1  in1 = 0; in2 = 1; cin = 0;
      ……
    end

    initial
      $monitor($time, " ns in1 = %b, in2 = %b, cin = %b, sum = %b, cout  = %b", in1, in2,
        cin, sum, cout);

endmodule
```

```
initial begin
   $dumpfile("test_fadd.vcd");
   $dumpvars;
end
```

# Instantiating the module

## Example: 4-bit full adder circuit



# Instantiating the module

```
module fadd4(in1, in2, cin, sum,cout);
 input[3:0] in1,in2;
 input  cin;
 output[3:0] sum;
 output cout;
 wire[2:0] temp_cout;

 fadd(in1[0],in2[0],cin,sum[0],temp_cout[0]);
 fadd(in1[1],in2[1],temp_cout[0],sum[1],temp_cout[1]);
 fadd(in1[2],in2[2],temp_cout[1],sum[2],temp_cout[2]);
 fadd(in1[3],in2[3],temp_cout[2],sum[3],cout);

 endmodule
```

# Data Objects in Verilog

- **Nets**
  - Used to interconnect modules and primitives
  - The following net types supported in verilog

| Net Type | Description |
|---|---|
| wire | Default net type |
| tri | Another name for wire |
| wand | Wired AND |
| triand | Another name for wand |
| wor | Wired OR |
| trior | Another name for wor |
| tri1 | Wire with built-in pullup |
| tri0 | Wire with built-in pulldown |
| supply1 | Always 1 |
| supply0 | Always 0 |
| trireg | Storage node for switch level modeling |

## Nets

- Default width of net object is 1-bit
  - Net objects can have width more than 1-bit

Example

```
wire a,b,c;      /* width = 1bit    */
wire[7:0] a      /* width = 8 bits */
wor[2:0] c;      /* width = 3 bits */
supply1 vcc;  /* width = 1 bit   */
supply0 gnd   /* width = 1 bit   */
```

## Registers

- Registers are used for modeling in procedural block.
- Registers can be 1-bit wide or declared as vectors.
- Registers are unsigned.

## Examples

```
reg a,b,c;          /* 1-bit register*/
reg [15:0] a;  /* 16-bit register */
reg [0:3] r;    /* 4-bit register */
reg [15:12] a  /* refers to 4 most significant bits of a*/
```

## Memories

- Memories are arrays of registers.
- Memory declaration is similar to a reg declaration with the addition of the range of words in the memory

Example

```
reg [7:0] b[0:15];      //b is a memory of 16 8-bit words
```

# Integers

- At least 32-bits wide
- Signed
- Declared using the keyword *integer*

# Reals

- A real data type holds a floating point number in IEEE format
- At least 32-bit wide
- Declared using the keyword *real*

Examples

```
integer number, count;
real result, fraction;
```

Time

- Used to represent current simulation time.
- Width of time data object is 64-bits.
- Variables of type Time is declared using keyword time

Example

```
time t1,t2;
```

Parameters

- Parameters are run time constants
- Default size is 32-bits.
- Parameters with ranges can be declared
- Parameters may be strings
- Note: Even though parameters are run time constants their values can be updated at compilation time.

Examples

```
parameter[13:0] message = " Hello Verilog";
parameter size = 8;
parameter msb = size-1;
```

# Events

- They does not represent any real hardware
- They does not hold any value
- They are used to signal something has occurred
- Events can not be passed through ports.

Examples
event birth;
event ack, parity-error;

# Number representation

- In verilog numbers are represented using the following format

Radix

- ‘b or ‘B binay
- ‘d or ‘D decimal (Default)
- ‘h or ‘H Hexa decimal
- ‘o or ‘O Octal

Examples

4’b0111

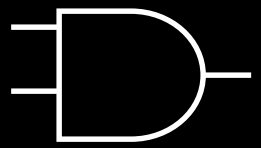
16’habcd

‘d1458

# 4-valued logic

- Verilog's nets and registers hold four-valued data
- 0, 1
  - Obvious
- Z
  - Output of an undriven tri-state driver
  - Models case where nothing is setting a wire's value
- X
  - Models when the simulator can't decide the value
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously
  - Output of a gate with Z inputs

# 4-valued logic



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

Output 0 if one input is 0

Output X if both inputs are gibberish

# System tasks

- System task commands are used to perform system related functions.
  - Examples
    - Printing messages
    - Reading and Writing files

- All system task commands begin with $ symbol.

## Display system task commands

- $dispaly
  - Can be used to printout binary, decimal, Hexadecimal or octal values.

Syntax

$display(|optional format specifier |, [value1],[value2],…);

Examples

```
$display (“%b %d”, num1, num2);
$display (“First data”);
$display (“Simulation time : %d”, sim_time);
```

- **$write**
  - Same as $display except that $display put new line character at the end of the results where as $write does not.

Examples

```
$write(“Verilog quick start”);
$write(“%h %o”, hex_num, oct_num);
```

- **$strobe**

  - Used when you want to print the values when all are changed at the current time unit.

Example
$strobe(“The new values are %h, %d “, hex_val, dec_val);

- **$monitor**

  Prints the data as they change.

Example
$monitor (“reset time = %d”, reset);

## System task commands for file operations

- $fopen
- $fdisplay
- $fwrite
- $fmonitor
- $fstrobe
- $fclose

# Operators

- Arithmetic Operators
  +, -, *, /, %
- Bitwise Operators
  | , &, ^, <<, >>
- Logical Operators
  &&, ||
- Negation operators
  ~, !
- Reduction Operators
  &, |, ^, ~&, ~|, ~^
- Concatenation operator
  { }
- Ternary operator
  Essentially an if-then-else statement.
  Uses two operators - ?, :

```
module test_op();

  reg   [3:0] a = 4'b1010;
  wire b;
  wire [3:0] c;

 assign b = !a;
 assign c = ~a;

 initial begin
   #10;
   $display("Logical: %b", b);
   $display("Bitwise: %b", c);
end

endmodule
```

Example

Result = a ? b : c;

The first operand (a) is logically evaluated. If true second operand(b) is returned otherwise third opearnd(c) is returned.

# Structural Modeling

- The module is implemented by connecting set of primitives defined by the language (called built-in primitives) or by the user (called user defined primitives)

- Verilog has a set of twenty six built-in primitives classified into 3 different categories
  - Logic Gates

    and or xor nand nor xnor

  - Buffers

    buf bufif0 bufif1 notif0
    notif1 pulldown pullup

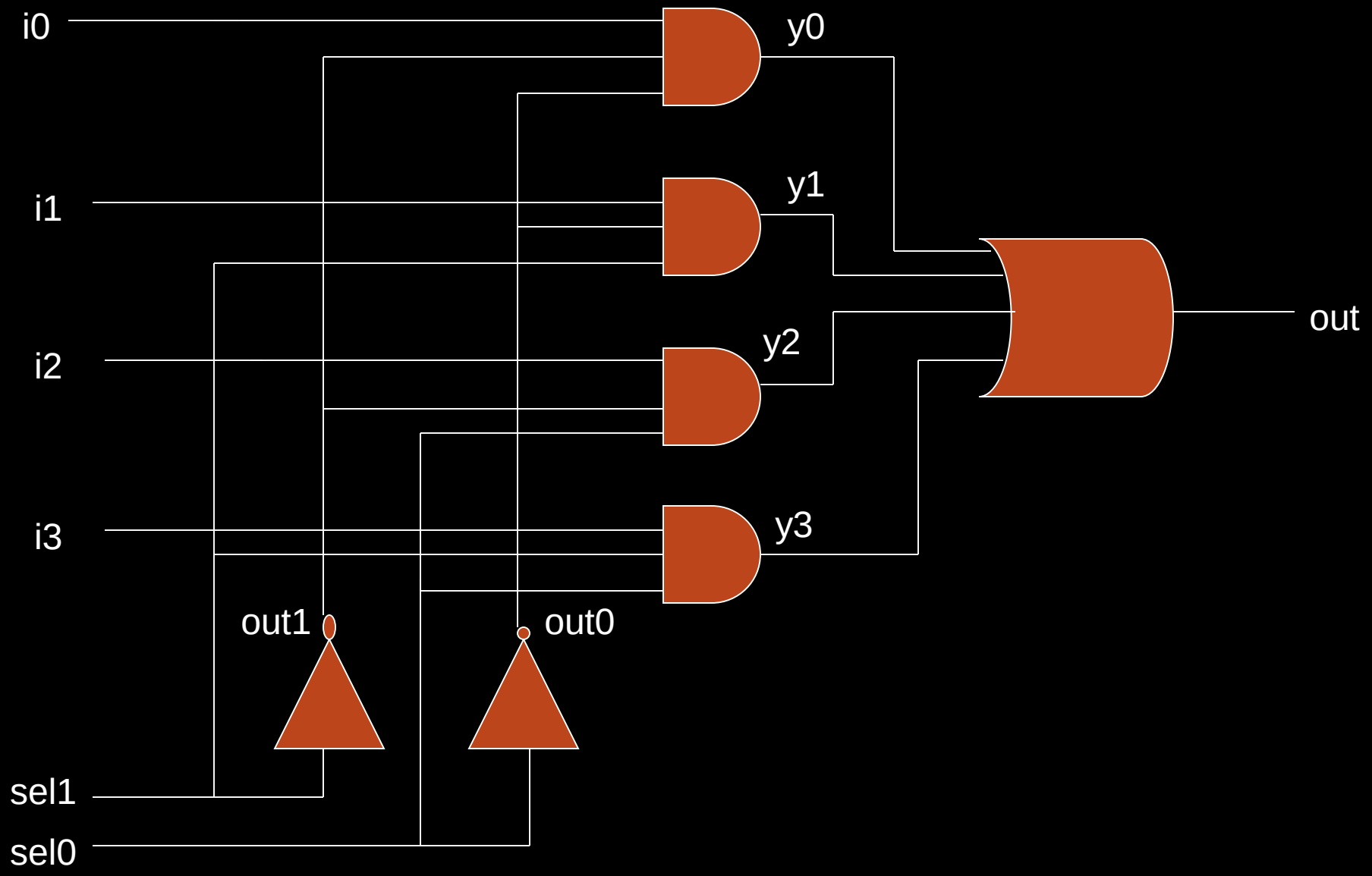
  - Transistor

    nmos pmos cmos rnmos
    rpmos rcmos tran tranif0
    tranif1 rtran rtranif0 rtranif1

# Gate Level Modeling

- A logic circuit can be designed by use of logic gates
- Verilog supports basic logic gates as pre defined primitives
- These primitives can be instantiated like modules except that they do not need a module definition.
- Verilog has two classes of basic gates
  - **and/or**
    - Have one scalar output and multiple scalar inputs.
    - The first terminal in the port list is output and rest are input terminals.
  - **buf/not**
    - Have one scalar input and one or more scalar outputs.
      - Last terminal in the port list is connected to the input and others are connected to the output.

# Example : 4_to_1 Multiplexer



```
module mux4_to_1(i0,i1,i2,i3,sel0,sel1, out);
        output out;                  //output port
        input i0,i1,i2,i3;           //input ports
        input sel0,sel1;            //control lines (input ports)
        wire out0, out1;           //internal wire declarations
        wire y0, y1,y2,y3;

        not(out0,sel0);
        not(out1,sel1);
        and(y0,i0,out0,out1);
        and(y1,i1,out0,sel1);
        and(y2,i2,sel0,out1);
        and(y3,i3,sel0,sel1);
        or(out,y0,y1,y2,y3);

endmodule
```

```
// Code your testbench here
// or browse Examples
module testbench;
  reg i0,i1,i2,i3, sel0,sel1;
  wire out;
  mux4_to_1 dut(.*);

  initial begin
   i0 = 1'b1; i1 = 1'b0; i2 = 1'b1; i3 = 1'b1; sel0 = 1'b0; sel1 = 1'b0;
   #10 i0 = 1'b1; i1 = 1'b0; i2 = 1'b1; i3 = 1'b1; sel0 = 1'b0; sel1 = 1'b1;
   #10 i0 = 1'b1; i1 = 1'b0; i2 = 1'b1; i3 = 1'b1; sel0 = 1'b1; sel1 = 1'b0;
   #10 i0 = 1'b1; i1 = 1'b0; i2 = 1'b1; i3 = 1'b0; sel0 = 1'b1; sel1 = 1'b1;
   end

initial begin
    $monitor ("out = %b", out);
    $dumpfile("testbench.vcd");
    $dumpvars;
    #50 $finish;
  end
endmodule
```

# User defined primitives

- User defined primitives describes a piece of logic with a truth table
- UDPs can be either combinational or sequential
- UDPs can have only one output and none of its inputs and outputs can be a vectors.

Syntax

```
primitive primitive_name(port id's);
output port_id_names;
input port_id_names;

table
    inputs  : output
endtable

endprimitives
```

Example: combinational UDPs (mux_2_to 1)

```
primitive mux2to1(y,sel,in1,in2);
output y;
input sel,in1,in2;

table
  //   sel    in1   in2   :  y
       0      0     ?     :  0;
       0      1     ?     :  1;       // "?" Can be 0,1 or x
       1      ?     0     :  0;
       1      ?     1     :  1;
       x      0     0     :  0;         // "x" unknown
       x      1     1     :  1;
endtable

endprimitive
```

# Data flow modeling

- In data flow model, the design in terms of the data flow between registers and how a design processes the data.
- In data flow model we do not instantiate any primitives.
- In Verilog we use continuous assignment statement to represent the data flow model.
- Delays may be associated with continuous assignment statement

## *Continuous assignment statement*

Syntax

assign <delay> <continuous_assign_var> = <list of assignments>;

- Left hand side of the expression must always be scalar or vector net or concatenation of scalar and vector net.
- The operands on the right hand side can be registers or nets or function calls.
- Registers and nets can be scalars or vectors.
- Note: Continuous assignment statements are always active.

Example mux2to1

```
module mux21(out, sel, inp1,inp2);
 output out;
 input sel, inp1, inp2;

     assign out = sel ? inp1 : inp2;

endmodule
```

Example mux4to1

```
module mux_4_to_1(out, i0,i1,i2,i3,s1,s0);
 output out;
 input i0,i1,i2,i3,s0,s1;
    assign out = (~s1 & ~s2 & i0) |
                 (~s1 & s0 & i1) |
                 (s1 & ~s0 & i2) |
                 (s1 & s2 & i3);
endmodule
```

# Behavioral Modeling

## Initial and always Block

- Basic components for behavioral modeling

Syntax of initial block

```
initial
begin
    … imperative statements …
end
```

- Runs when simulation starts
- Terminates when control reaches the end
- Good for providing stimulus

Syntax of always block

```
always @(sensitivity variables)
  begin
    … imperative statements …
  end
```

- Runs when simulation starts
- Restarts when control reaches the end
- Good for modeling/specifying hardware

- Both ***always*** and **initial** statements are concurrent statements.
- Statements within the body of the ***always*** and ***initial*** block may be **sequential** or **concurrent**.
- Verilog uses ***begin .. end*** block for **sequential** modeling and ***fork .. join*** block for **concurrent** modeling.

Example

```
module test;
initial
  begin
    #3 $display($time, " Verilog HDL");
    #3 $display($time, " VHDL HDL ");
    #1 $finish;
  end
 always
   fork
      #2 $display($time, " SystemC");
      #1 $display($time, " Simulation");
  join
endmodule
```

Results

```
1 Simulation
2 SystemC
3 Verilog HDL
3 Simulation
4 SystemC
5 Simulation
6 VHDL HDL
6 SystemC
```

# Procedural Assignment

- The value placed on a variable remain unchanged until another procedural assignment updates the variable with different value.
- Syntax

<assignment>

::= <lvalue> = <expression>

- Two types of procedural assignment statements
  - Blocking assignment
  - Nonblocking assignment

# Blocking Assignment

- The **= operator** is used to specify blocking statement
- RHS has more bits than the LHS, then MSB is truncated
- RHS has fewer bits than the LHS, then MSB is padded with Zeros

# Blocking Assignment - Example

```
initial
begin
   x = 0; y= 1; z = 1;
   count = 0;
   reg_a = 16'b0;
   reg_b = reg_a;
   #15 reg_a[2] = 1'b1;
   #10 reg_b[15:13] = {x, y, z}
   count = count +1;
end
```

Time = 0

x = 0 to statement

reg_b = reg_a

Time = 15

reg_a[2] = 0

Time = 25

reg_b[15:13] = {x, y, z}

count = count + 1

# Nonblocking Assignment Example

The **<= operator** is used to specify nonblocking statement

```
initial
begin
   x = 0; y= 1; z = 1;
   count = 0;
   reg_a = 16'b0;
   reg_b = reg_a;
   #15 reg_a[2] <= 1'b1;
   #10 reg_b[15:13] <= {x, y, z}
   count <= count +1;
end
```

Time = 0

x = 0 to statement

reg_b = reg_a &

count = count + 1

Time = 10

reg_b[15:13] = {x, y, z}

Time = 15

reg_a[2] = 0

# Application of Nonblocking Assignments

```
always @(negedge clk)
begin
    a <= #2 x;
    b <= #1 (p & q);
   d <= #3 a; //old value of a
    variable
end
```

- A read operation is performed on all the RHS variables, x, p, q, m , n, a at negative edge of the clock
  - They are evaluated and stored in temporary variables by the simulator
- The write operation to LHS is done after the specified time delay
  - The order in which each statements are written – not important, the assignment happens concurrently

# Procedural Assignment Example

## Blocking statement

```
always @ (posedge clock)
   a = b;

always @ (posedge clock)
   b = a;
```

## Nonblocking statement

```
always @ (posedge clock)
   a <= b;

always @ (posedge clock)
   b <= a;
```