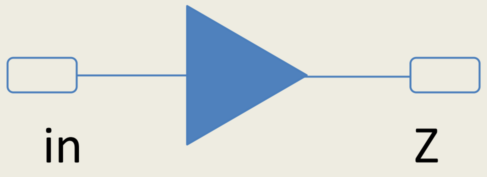
VERILOG-BASICS

How to write a basic Verilog code and test bench?

-Language constructs

Basic building blocks: modules

- A design in Verilog is modeled using the concept of "module".
- A module will have two parts
  - *A port declaration part*
  - *A functional description part*
- Module declarations cannot be nested.
- Modules are:
  - *Declared*
  - *Instantiated*
- A module may contain any of the following items:
  - *data_type_declarations*
  - *parameter_declarations*
  - *module_instances*
  - *primitive_instances*
  - *generate_blocks*
  - *procedural_blocks*
  - *continuous_assignments*
  - *task_definitions*
  - *function_definitions*
  - *specify_blocks*



ANSI C style port list

```
module buffer (output Z, input in);
    assign #1 Z = in;
endmodule
```

Old style port list

```
module buffer (Z,  in);
  output Z;
  input in;
    assign #1 Z = in;
endmodule
```

Ports and Comments

- Module interface is defined using ports. Each port must be explicitly declared as one of the following types:
  - *input*
  - *output*
  - *inout*
- Ports can be
  - *Scalar (single bit) - don't specify a size*

    input cin;
  - *Vector (multiple bits) - specify size using range*

    Range is MSB to LSB (left to right)

    Don't have to include zero (D[2:1])

    output [7:0] OUT;

    input [0:4]IN;

- Verilog supports 2 type of comment syntaxes
  - *Single line comment start with //, and end with newline.*
  - *Block comment, start with /*, and end with */.Block comment cannot be nested.*
- Example

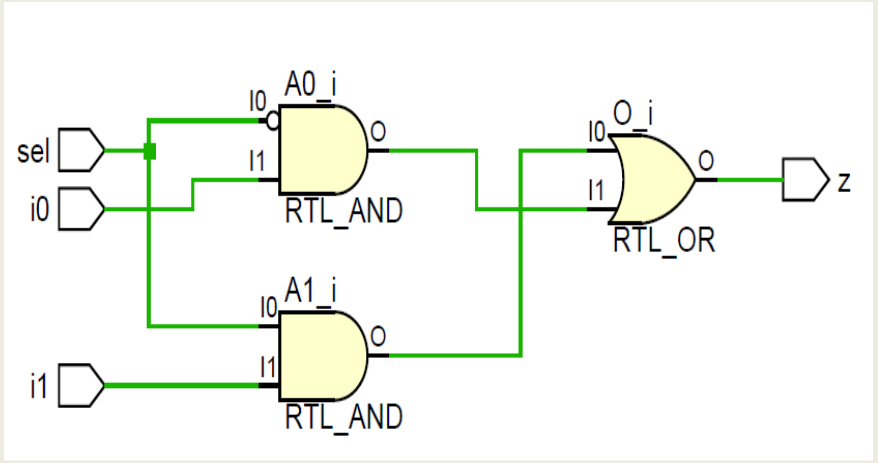
```
/* This is how you comment out a
number of lines in Verilog. Can be used to
intricately describe a code. The comment
stops where it encounters this => */
    input status; // 0:ready, 1:not ready
    output data; // sync with clock mClock
```

Levels of Modeling

- Structural
  - *Use primitives and lower-level module instantiations.*
  - *Logic gates and Verilog primitives*
  - *Very close to physical implementation.*
  - *Easy (straightforward) synthesis.*
- Dataflow
  - Define output signals in terms of input signal transformations.
- Behavioral
  - *Specify algorithmically the expected behavior of the system.*
  - *Close to natural language description.*
  - *Most difficult to synthesize.*
  - *Best for verification.*

Structural Modeling

- Structural elements of a Verilog structural description are logic gates and user-defined components connected by wires
- Structural description can be viewed as a simple netlist composed of nets that connect instantiations of gates.
- Primitives are pre-defined gates already described in the Verilog language.
- Advantages:
  - *Hand-Designed systems always the most efficient*
  - *Power, area, frequency requirement can be most accurately met by designer.*
  - *Existence of proven sub-blocks*
  - *IP-Cores readily available*
- Disadvantages
  - *This kind of modeling can get tedious for very large designs*
  - *Architecture / hardware of problem being solved may not be known always*
  - *Arriving at structure first and then modeling it will take lots of time*
  - *Building edge triggered circuits with structural modeling alone can be very tricky*
- Though being ideally the best way to model a hardware time-to-market issues and complexity involved works against it.



```
//2-to-1 Multiplexer
module mux2(
  output z,
  input i0, i1, sel );
    wire nsel,a0,a1;
    not  inv (nsel, sel);
   and A0(a0, nsel, i0);
   and A1(a1,sel,i1);
   or O(z,a1,a0);
endmodule
```

Using primitives

- Pre-defined gates already described in the Verilog language
- Can have only one output but any number of inputs. They need not be declared, they are only called.
- First variable in port list of primitive is always output and naming of primitives is optional.
- Language has no restriction on inputs but depends on technology being targeted to.
- Delays can be specified again optionally.
- Any number of gates of same functionality can be called in single line.
- Basic combinational gates are available as primitives.
- Used directly in the code as smaller case key-words.
- Some examples :and, nand, or, nor, not, xor, xnor.

```
//1-bit Adder
module adder(
  output sum, carry,
  input a, b, cin);
   xor XO (sum,a,b,cin);
   and A0(a0, a,b),
          A1(a1,b,cin),
          A2(a2,cin,a);
   or O(carry,a0,a1,a2);
endmodule
```

Component Instantiation

- Instantiation
  - *Predefined modules can be called just like primitives*
  - *An Instance is a unique copy of a pre-defined module.*
  - *Predefined modules are connected to one another by means of nets.*
- Component instantiation can be done in two ways
  - *Named port connection / calling by reference.*
  - *Ordered port connection / calling by position.*
- Most simulators /synthesis tools require components to be compulsory named
- Component instantiations do not accommodate delays like primitives.
- While simulating component should have been compiled before it can be called in a top module.

```
// 4-to-1 multiplexer
module mux4(
 input d0,d1,d2,d3,
 input [1:0] sel,
 output z);
  wire z1,z2;
/*instances must have unique names within
current module. Connections are made using
.portname(expression) syntax.*/
// order doesn't matter…
  mux2 m1(.sel(sel[0]),.i0(d0),.i1(d1),.z(z1));
  mux2 m2(.sel(sel[0]),.i0(d2),.i1(d3),.z(z2));
  mux2 m3(.sel(sel[1]),.i0(z1),.i1(z2),.z(z));
/*could also write "mux2 m3(z,z1,z2,sel[1])"
NOT A GOOD IDEA! */
endmodule
```

Dataflow Modeling

- Dataflow modeling consists of continuous assignment statements
  - *They start with the keyword 'assign' . These statements are always active.*
  - *They are executed whenever there is a change in a variable on the right hand side of the statement.*
  - *They are used when the system being designed can be completely represented in Boolean equation format.*
  - *Several assignment statements are concurrent in nature(they execute in parallel).*
  - *They can also represent the propagation delay from input to output.*
- This type of execution model is called "**dataflow**" since evaluations are triggered by data values flowing through the network of wires and operators.

```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(
 input i0,i1,sel,
 output z,zbar);
// again order doesn't matter (concurrent execution!)
// syntax is "assign LHS = RHS" where LHS is a wire/bus
// and RHS is an expression
   assign z = sel ? i1 : i0;
   assign zbar = ~z;
endmodule
```

Vectors and Parameters

- Parameters are run time constants
  - *Are declared within a module*
  - *Their scope lies within the module*
  - *Can be overwritten during component instantiation*
  - *Used to make a code scalable*
  - *Module can have more than one parameter*

- Signals which are multi-bits are declared as vectors
  - *Can be declared as [high:low] or [low:high]*
  - *Common usage is* ***[MSB:LSB]*** *where MSB > LSB; usually LSB is 0.*
  - *A bit of the vector is accessible.*
  - *Examples:*

```
wire [n:0]  carry_wires // vector being declared
carry_wires[5:2] = 4'd1010 // part of vector being
                                     //assigned value
 outp = carry_wires[5:3] // part of vector being
                                     //accessed
```

```
// 2-to-1 multiplexer, W-bit data
module mux2 #(parameter W=1) // data width, default 1 bit
( input [W-1:0] i0,i1,
  input sel,
  output [W-1:0] z );
  assign z = sel ? i1 : i0;
endmodule

// 4-to-1 multiplexer, W-bit data
module mux4 #(parameter W=2) // data width, default 2 bit
( input [W-1:0] d0,d1,d2,d3,
  input [1:0] sel,
  output [W-1:0] z);
  wire [W-1:0] z1,z2;
   mux2 #(.W(W)) m1(.sel(sel[0]),.i0(d0),.i1(d1),.z(z1));
   mux2 #(.W(W)) m2(.sel(sel[0]),.i0(d2),.i1(d3),.z(z2));
   mux2 #(.W(W)) m3(.sel(sel[1]),.i0(z1),.i1(z2),.z(z));
endmodule
```

Language conventions and semantics

Conventions:

- Verilog is case-sensitive
- Some simulators are case-insensitive
- Advice: - Don’t use case-sensitive feature!
- Keywords are always lower case
- Different names must be used for different items within the same scope
- Identifiers can have
  - ✓ *Upper case and lower case alphabets*
  - ✓ *Decimal Digits (but should not start with digits)*
  - ✓ *Underscore (again not at the beginning*

Semantics:

- All statements are terminated with ‘;’
- Comments :
  - ✓ *all characters after ‘//’ in a line are treated as comments*
  - ✓ *multiple line comments begin and end with ‘*’*
- Compiler directives begin with ‘`’
- Built in system task and functions begin with ‘$’
- Strings enclosed with double quotes must be on one line

Four valued logic

Since we're describing hardware, we'll need to represent the values that can appear on wires. Verilog uses a 4-valued logic. A single bit can have one of FOUR values:

❑ "0"→ Numeric 0, logical FALSE, "low"
❑ "1"→ Numeric 1, logical TRUE, "high"
❑ "x"→ Unknown or ambiguous value
❑ "z"→ No value (high impedance)

Why **x**?
– Could be a conflict, could be lack of initialization
Why **z**?
– Nothing driving the signal, Tri-states

Verilog also has the notion of "drive strength" but we can safely ignore this feature for our purposes.
• Multiple strong conflicting drivers => short circuit
• Weak signals => circuit can't operate, unexpected results

❖ In Simulation
– Can detect x or z using special comparison operators
❖ In Reality
– Cannot detect x or z
– No actual 'x' – electrically just 0, 1, or z
– Except for some uninitialized signals, x is bad!

Number representation

- Representation: <size>'<base><number>
  - size => number of bits (regardless of base used)
  - base => base the given number is specified in
  - number => the actual value in the given base
- Can use different radix(base)
  - d or D : Decimal – default if no base specified!
  - h or H : Hexadecimal Hex
  - o or O : Octal
  - b or B : Binary
- Size defaults to at least 32.
  - You should specify the size explicitly!
  - Why create 32-bit register if you only need 5 bits?
  - May cause compilation errors on some compilers
- To be absolutely clear in your intent **it's usually best to explicitly specify the width and radix.**

Number	Decimal Equivalent	Actual Binary
4'd3	3	0011
8'ha	10	00001010
8'o26	22	00010110
5'b111	7	00111
8'b0101_1101	93	01011101
8'bx1101	-	xxxx1101
-8'd6	-6	11111010

Data types

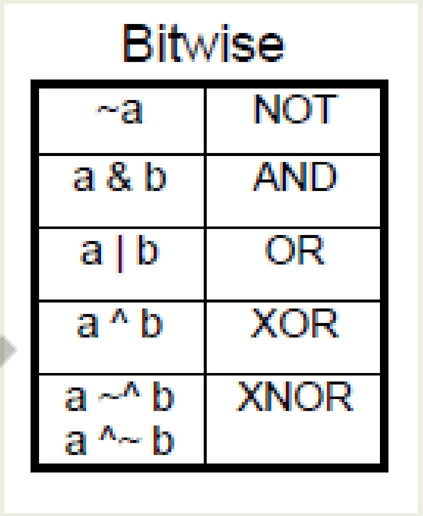
- Verilog has two major data types:
  Net and Variable(register)
- Nets
  - *Represent interconnects / physical connection from one point to other*
  - *Primarily used only in dataflow and structural modeling*
  - *Must be driven by a driver, such as a gate or continuous assignment statement*
  - *Default value is high impedance*
  - *Examples*
    - *wire, wand, wor*
    - *tri, triand, trior*
    - *supply0, supply1*
- Signed vs Unsigned
  - *Net types and reg variables unsigned by default*
  - *Have to declare them as signed if desired!*

    reg signed [7:0] signedreg;

    wire signed [3:0] signedwire;
- Variable
  - *Represent interconnects / physical connection from one point to other*
  - *Analogous to the concept of storage / holds some value*
  - *Primarily used only in behavioral modeling*
  - *There are various kinds*
    - *reg // stores a value, can be single bit or vector*
    - *integer // used in computations, is of 32 bits*
    - *time // used to see the simulation time*
    - *real // used to store real numbers, uses 64 bits*
    - *realtime // stores time as real numbers*
  - *Need not necessarily result in flip-flop/ latch*
  - *Default value is 'x'*
  - *Cannot be driven by a gate*

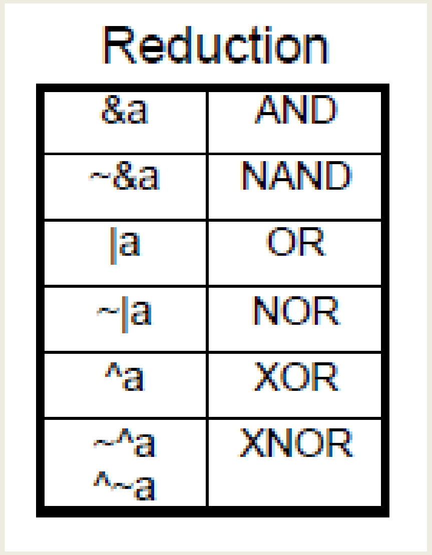
Operators

Operator Type	
Bitwise	~, &, |, ^, ~^, ^~,
Logical	!, &&, ||
Unary/Reduction	&, ~&, ^, ~^, ^~, |, ~|
Relational	<, <=, >, >=
[In]Equality	==, !=, ===, !==
Conditional	?:
Arithmetic	+, -, ** (power) , *, /, %, >>,<<,>>>,<<<
Concatenation	{}, {{}}



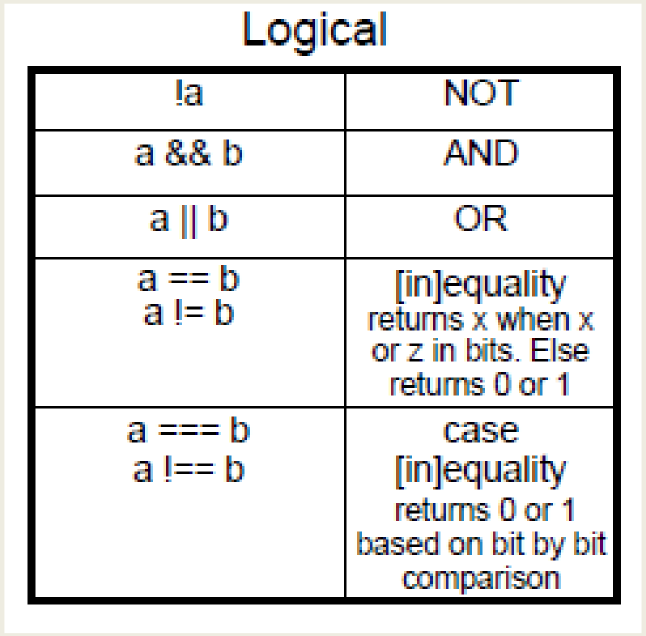
Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR



Reduction

&a	AND
~&a	NAND
|a	OR
~|a	NOR
^a	XOR
~^a ^~a	XNOR



Logical

!a	NOT
a && b	AND
a || b	OR
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Generate statement

- Simply replicates a given set of function for a specified number of time.
- Two types of generate statements available:
  - *For generate*
    - Variables used inside for generate loop known as *'genvar'*
    - Loop inside for generate should be labeled
  - *If generate*
    - Used for conditional use of hardware / logic
- Statement is concurrent in nature.

```
//Code for hierarchal design
module adder_bit_n #(parameter n = 8)
 ( output [n-1:0] sum,
  output carry_out,
  input [n-1:0] first, second,
  input carry_in );
   genvar j;
   wire [n:0] carry_wires;
   assign carry_wires[0] = carry_in;
    generate
     for (j=0; j <n; j = j+1)
      begin :series
       adder A(.A(first[j]),.Ci(carry_wires[j]),
             .B(second[j]), .Su(sum[j]),
             Co(carry_wires[j+1]));
      end
    endgenerate
   assign carry_out = carry_wires[n];
endmodule
```

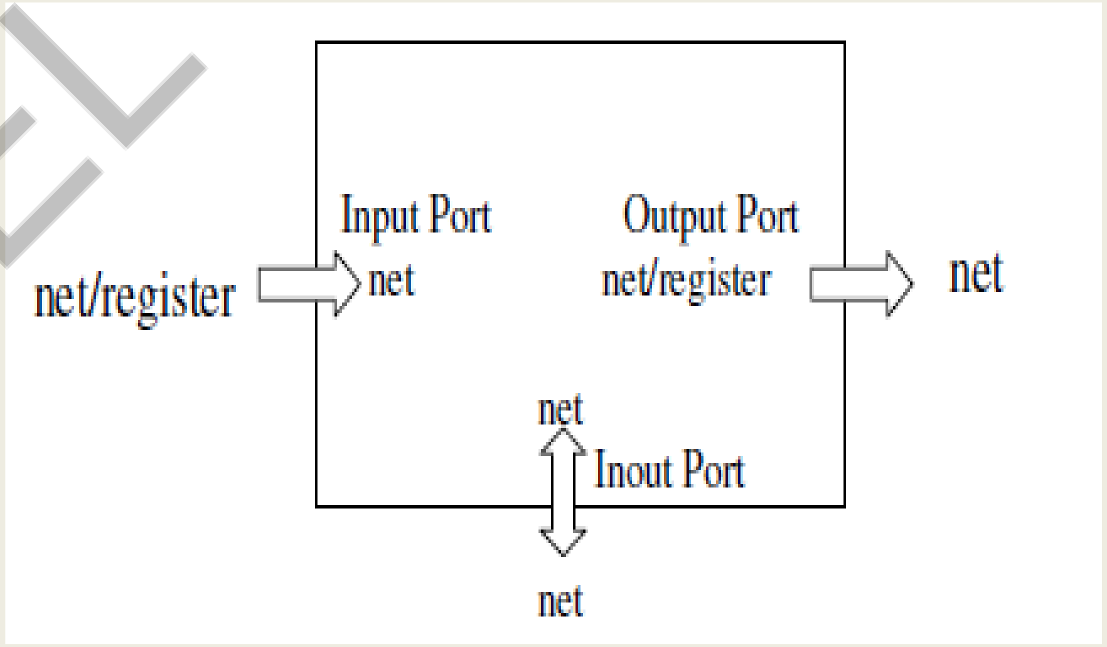
Behavioral Modeling

- Most advanced coding style which is flexible, high-level and closest to software programming languages.
- Behavior: Actions a circuit is supposed to perform when it is active.
- Present day problems are quite complex and so are the solutions. Functionality required is known but not hardware.
- Behavior of system can be very easily explained in higher level abstraction.
- Represents both concurrent and sequential behaviour of hardware
- Repetitive task can be expressed only once and then called many number of times
- Uses procedural blocks which are of two types:
  - *initial*
  - *always*
- Can have more than one procedural blocks. All procedural blocks work in parallel / concurrent in nature. Each procedural block contain set of statements which execute sequentially.

Ports and Datatypes

We can add the *reg* keyword to *output* or *inout* ports (we wouldn't be assigning values to *input* ports!), or we can declare nets using *reg* instead of *wire*.

We've been using wire declarations when naming nets (ports are declared as wires by default). However nets appearing on the LHS of assignment statements inside of *always* blocks **must** be declared as type reg.



Procedural blocks

Initial

- Used primarily to initialize variables and in testbenches.
- Terminates when control reaches the end.
- Will not synthesize into hardware.
- More than one statement in a block needs to be grouped with ‘*begin*’ & ‘*end*’
- Multiple initial blocks start execution in parallel at time 0 and end independent of other blocks.
- Will not have sensitivity list

Always

- Used to describe the functionality of a system
- Restarts when control reaches the end.
- Will synthesize into hardware as defined by statements written inside the block.
- More than one statement in a block needs to be grouped with ‘*begin*’ & ‘*end*’
- Multiple ‘always’ blocks execute in parallel
- May have a sensitivity list

Procedural block : Examples

```
initial
begin
 a = 3’b0;
 #10 clr = 1’b0;
 #21 clr = 1’b1;
end

initial
begin: test_loop
 integer i;
 for (i=0; i<=15; i=i+1;
  #5 test_in = i;
end
```

```
module comparator(
 output reg dgv, dlv, dev,
 input [4:0] data, value);
  always@(data, value)
  begin
   dgv = (data > value);
   dlv = (data < value);
   dev = (data == value);
  end
endmodule
```

Conditional statement

- Statements written inside procedural blocks are known as procedural assignment statements.
- There are two sub-classes known as
  - conditional assignment statements
    - if-else, case
  - loop statements
    - for, while, forever, repeat
- Some of these statements synthesize while others do not.
- Conditional statements are behavioral equivalent of the conditional operator.
  - These can be nested as many times as needed
  - When written properly will infer a multiplexer or mux like structure

```
// 4-to-1 multiplexer
module mux4(
 input a,b,c,d, input [1:0] sel, output
reg z,zbar);
 always @(*)
  begin
   if (sel == 2'b00) z = a;
   else if (sel == 2'b01) z = b;
   else if (sel == 2'b10) z = c;
   else if (sel == 2'b11) z = d;
   else z = 1'bx;
/* when sel is X or Z , statement order
matters inside always blocks  so the
following assignment happens *after*
the if statement has been evaluated*/
   zbar = ~z;
   end
endmodule
```

Case statement

- Chains of **if-then-else** statements aren’t the best way to indicate the intent to provide an alternative action for every possible control value.

- *case* looks for an exact bit-by-bit match of the value of the case expression (e.g., sel) against each case item, working through the items in the specified order. *casex*/*casez* statements treat X/Z values in the selectors as don’t cares when doing the matching that determines which clause will be executed.

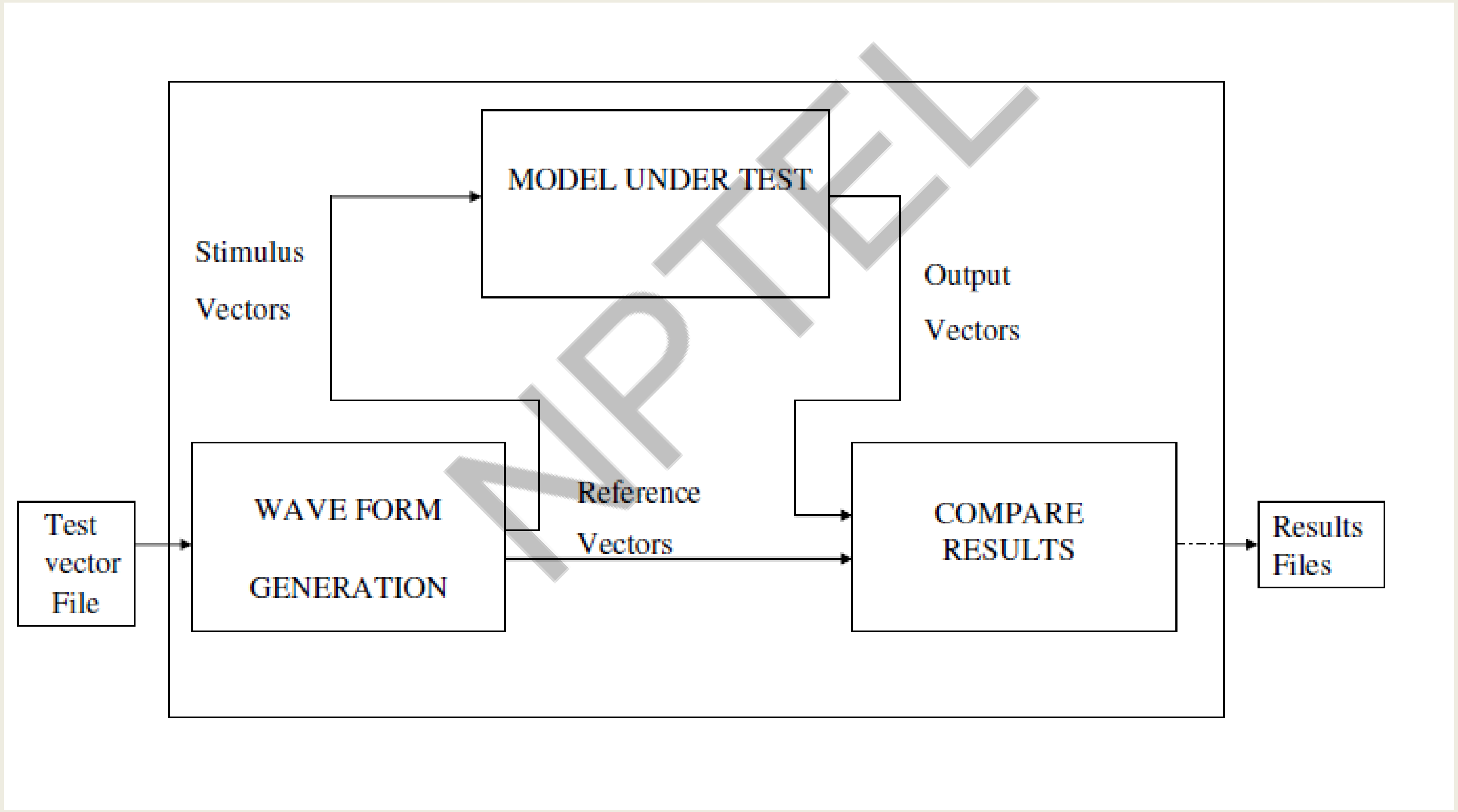
```
// 4-to-1 multiplexer
module mux4(
 input a,b,c,d, input [1:0] sel, output reg
z,zbar);
 always @(*)
  begin
   case (sel)
    2’b00: z = a;
    2’b01: z = b;
    2’b10: z = c;
    2’b11: z = d;
    default: z = 1’bx; // in case sel is X or Z
   endcase
   zbar = ~z;
  end
endmodule
```

Sensitivity list and Timing control

- An always block may or may not have any sensitivity list
- Sensitivity list controls when the block will be executed
- An always block without sensitivity list and no timing control statements will act like an infinite loop.
- Example
  - *always*
  - *always* @(*) → blocks are evaluated whenever any value used inside changes.
  - Equivalently we could have written *always* @(a, b, c, d, sel) *begin* … *end*

- Timing control will stop/ delay the simulator from executing the remaining piece of code until some condition is true
- There are 2 kinds of timing controls
  - Delay Based (Postpones the simulation for some time)
    - Inter assignment delay,
    - Intra assignment delay
  - Event Based (Suspends the simulation for some time)
    - always@(data, value)
    - wait(enable)

Test bench structure



Components of test bench

- The simplest test benches are those that apply some sequence of inputs to the circuit being tested so that its operation can be observed in simulation.
- Waveforms are typically used to represent the values of signals in the design at various points in time
- A test bench must consist of a component declaration corresponding to the unit under test(UUT), and a description of the input stimulus being applied to the UUT.
- A test bench is a top level module without inputs and outputs
  - Data type declaration
    Declare storage elements to store the test patterns
  - Module instantiation
    Instantiate pre-defined modules in current scope
  - Applying stimulus
    Describe stimulus by behavior modeling
    Stimuli are given by variables of data type register
  - Display results
    Response is captured by variables of data type net
    By text output, graphic output, or waveform display tools
- Three kinds of testbenches are in use
  - One which only supplies the stimuli
  - One which not only supplies stimuli but also captures the result
  - One which supplies stimuli, captures result and also carries out timing check

Test bench example

```
module full_adder
( input a, b, cin,
output reg sum, cout);
always @(a or b or cin)
begin
sum = a ^ b ^ cin;
cout = (a & b) | (a & cin) | (b & cin);
end
endmodule
```

```
module full_adder_4bit
( input [3:0] a, b,
input cin,
output [3:0] sum,
output cout),
wire c1, c2, c3;
// instantiate 1-bit adders
full_adder FA0(a[0],b[0], cin, sum[0], c1);
full_adder FA1(a[1],b[1], c1, sum[1], c2);
full_adder FA2(a[2],b[2], c2, sum[2], c3);
full_adder FA3(a[3],b[3], c3, sum[3],
cout);
endmodule
```

```
module test_adder;
reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;
full_adder_4bit dut(a, b, cin,sum, cout);
initial
begin
a = 4'b0000;
b = 4'b0000;
cin = 1'b0;
#50;
a = 4'b0101;
b = 4'b1010;
// sum = 1111, cout = 0
#50;
a = 4'b1111;
b = 4'b0001;
// sum = 0000, cout = 1
#50;
a = 4'b0000;
b = 4'b1111;
cin = 1'b1;
// sum = 0000, cout = 1
#50;
a = 4'b0110;
b = 4'b0001;
// sum = 1000, cout = 0
end // initial begin
endmodule // test_adder
```

Compiler directives

- Macro definitions: `define
- Conditional compilation: `ifdef, …
- Include other source files: `include
- Control simulation time units: `timescale
- No implicit net declarations: `default_nettype none

References

- Slide source1
- Slide source2
- IEEE Verilog 2001 Standard
- Comparision of HDLs