

What is Hardware Description Language (HDL)?

- Allows designer to specify logic function only. Then a computer-aided design (CAD) tool produces or synthesizes the optimized gates.
- Most commercial designs built using HDLs
- Two leading HDLs:
 - Verilog
 - ▶ Developed in 1984 by Gateway Design Automation
 - ▶ Became an IEEE standard in 1995
 - VHDL
 - ▶ Developed in 1981 by the Department of Defence
 - ▶ Became an IEEE standard in 1987

Why HDLs?

- System specification is behavioral
- Manual Translation of design in Boolean equations
- Handling of large Complex Designs
- Hardware characteristics (viz., connections of parts, concurrent operations, concept of propagation delay and timing) cannot be captured by traditional programming languages

The Role of HDL

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction
- Interoperability: models at multiple levels of abstraction
- Technology independence: portable model
- Design re-use and rapid prototyping

Verilog Vs. VHDL

■ Verilog HDL

- Has very good acceptance in ASIC, particularly lower level designs (register transfer level and below)
- Results in fast simulations
- Relatively simple, easy in first contacts
- Mostly used in North America and Japan, especially among industrial supporters

■ VHDL

- Relatively weaker in lower level designs, but superior in higher and system level designs
- Results in slower simulations
- Very flexible, but also difficult
- Very popular in academia
- Used in Europe, significant number also in US and Canada
- Disliked in Japan, but gaining popularity worldwide

HDL to Gates

■ Simulation

- Input values are applied to the circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware

■ Synthesis

- Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

Definition of Module

- Interface: port and parameter declaration
- Body: Internal part of module

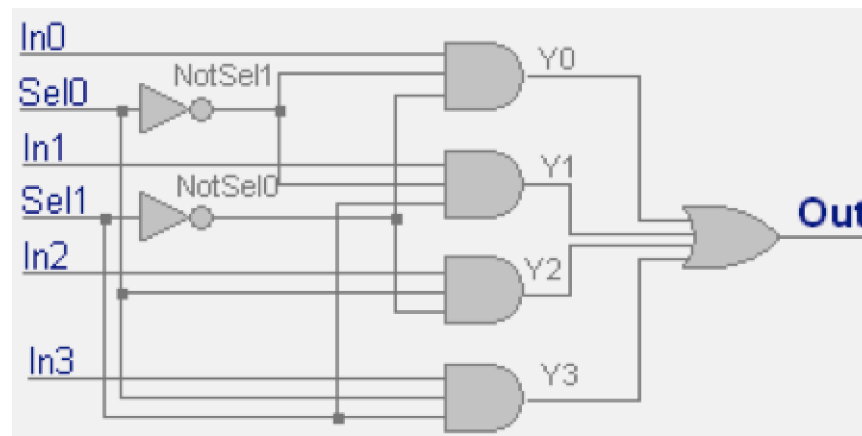
```
module < module_Name > (< Output_Input_Port_List >);  
    Input_Port_Declarations;  
    Output_Port_Declarations;  
    Internal_part_of_module;  
endmodule
```

Verilog Modules

- Two types of Modules:
 - **Behavioral:** Describe what a module does
 - **Structural:** Describe how a module is built from simpler modules

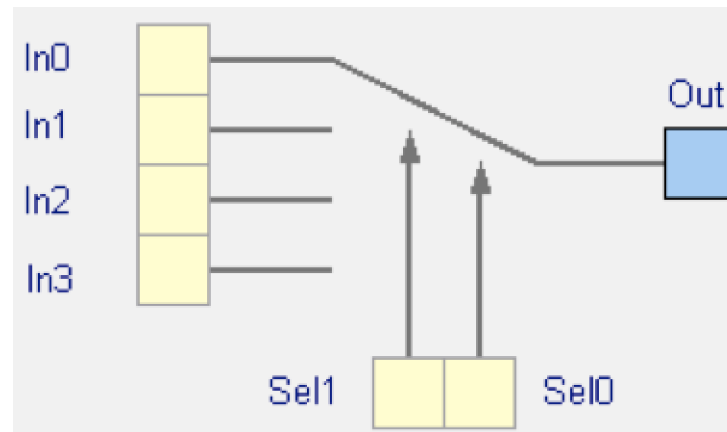
Structural Module

- When Verilog was first developed (1984) most logic simulators operated on *netlists*
- Netlist: list of gates and how they're connected
- A circuit is specified in terms of instantiations of lower level components (i.e., logic gates, which are Verilog primitives) connected with internal signals.
- Translation into a physical circuit is straightforward.

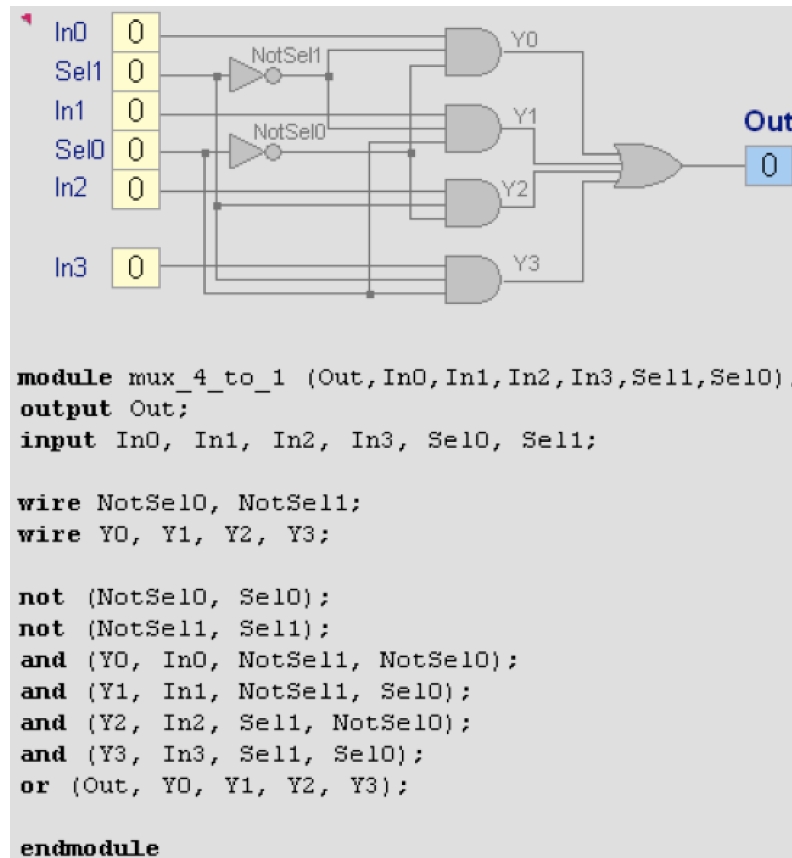


Behavioral Module

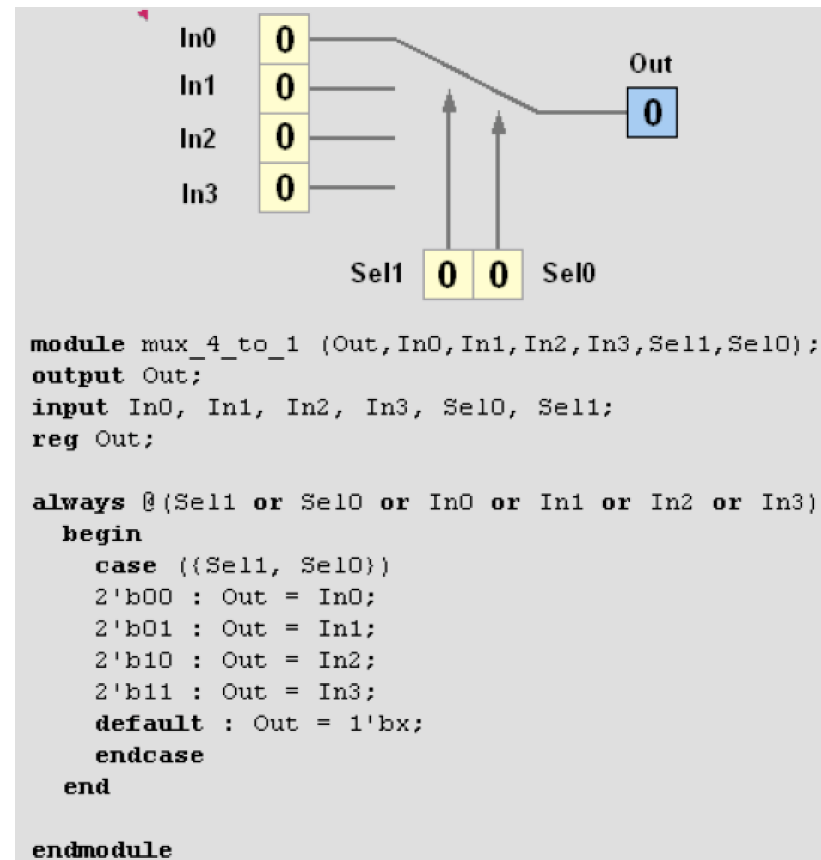
- A circuit is specified in terms of expected behavior
- Easier to write, Simulates faster, More flexible
- Closest to a natural language description of the circuit functionality, but also the most difficult to synthesize



Structural Module: Verilog Code



Behavioral Module: Verilog Code

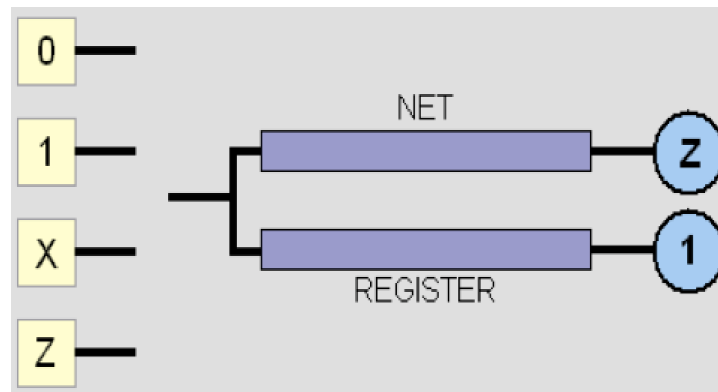


Two Main Data Types

- Nets represent physical connections between hardware elements
 - Do not hold their value
 - Take their value from a driver such as a gate or other module
 - Cannot be assigned in an *initial* or *always* block
- Regs represent data storage even if disconnected
 - Behave exactly like memory in a computer
 - Hold their value until explicitly assigned in an *initial* or *always* block
 - Never connected to something
 - Can be used to model latches, flip-flops, etc., but do not correspond exactly
 - Shared variables with all their attendant problems

Two Main Data Types

- The main difference between *nets* and *registers* can be observed when they are disconnected from the source:
 - *nets* switch to high-impedance
 - *registers* preserve their previous values

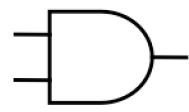


Four-valued Data

- 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

Four-valued Logic

- Logical operators work on three-valued logic



	0	1	X	Z	
0	0	0	0	0	← Output 0 if one input is 0
1	0	1	X	X	
X	0	X	X	X	← Output X if both inputs are gibberish
Z	0	X	X	X	

Nets and Registers

■ Wires and registers can be bits, vectors, and arrays

- **wire** x; //Simple wire
- **tri** [15 : 0] databus; // 16-bit tristate bus
- **tri** #(5, 4, 8) y; // Wire with delay
- **reg** [-1 : 4] z; // Six-bit register
- **integer** array_interger[0 : 1023]; // Array of 1024 integers
- **reg** [31 : 0] mem[0:63]; // A 32-bit memory

Gate-level Primitives

- Verilog provides the following:
 - **and** **nand** logical AND/NAND
 - **or** **nor** logical OR/NOR
 - **xor** **xnor** logical XOR/XNOR
 - **buf** **not** buffer/inverter
 - **bufif0** **notif0** Tristate with low enable
 - **bifif1** **notif1** Tristate with high enable

User-Defined Primitives

- Way to define gates and sequential elements using a truth table
- Often simulate faster than using expressions, collections of primitive gates, etc.
- Gives more control over behavior with X inputs
- Most often used for specifying custom gate libraries

User-Defined Primitives: An OR Primitive

```
primitive MY_OR(Z, X, Y);  
  output Z;  
  input X, Y;  
  table  
    00 : 0;  
    01 : 1;  
    10 : 1;  
    11 : 1;  
  endtable  
endprimitive
```

- Truth table may include don't-care (?) entries
- for $X = 1$, $Y = ?$ and $Z = 1$ can be written as $1? : 1$

User-Defined Primitives: A Sequential Primitive

```
primitive dff(q, clk, data);
output q; reg q;
input clk, data;
table
    // clk data q new-q
    (01) 0 : ? : 0; // Latch a 0
    (01) 1 : ? : 1; // Latch a 1
    (0x) 1 : 1 : 1; // Hold when d and q both 1
    (0x) 0 : 0 : 0; // Hold when d and q both 0
    (?0) ? : ? : -; // Hold when clk falls
    (??) ? : ? : -; // Hold when clk stable
endtable
endprimitive
```

Continuous Assignment

- Another way to describe combinational function
- Convenient for logical or datapath specifications

```
wire [8:0] sum; //Define bus width
```

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

```
wire carryin;
```

```
// Continuous assignment: permanently sets the value
```

```
// of sum to be  $a + b + \text{carryin}$ 
```

```
// Recomputed when a, b, or carryin changes
```

```
assign sum =  $a + b + \text{carryin}$ ;
```

Initial and Always Blocks

- Basic components for behavioral modeling

initial

begin

<imperative statements>
<imperative statements>
<imperative statements>

end

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

always

begin

<imperative statements>
<imperative statements>
<imperative statements>

end

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

Initial and Always

- Run until they encounter a delay

```
initial begin  
    #10 a = 1; b = 0;  
    #10 a = 0; b = 1;  
end
```

- or a wait for an event

```
always @(posedge clk) q = d;  
always begin  
    wait(i);  
        a = 0;  
    wait(~i);  
        a = 1;  
end
```

Procedural Assignment

- Inside an **initial** or **always** block:
$$\text{sum} = a + b + \text{cin};$$
- Just like in C: RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
 - Two possible sources for data
- LHS must be a reg
 - Primitives or continuous assignment may set wire values

Control Statements

- **if, else, repeat, while, for, case** - it's Verilog that looks exactly like C
- Imperative Statements

```
if (Condition) begin  
.....  
end  
else begin  
.....  
end
```

```
case (Option)  
Option_1 : .....;  
Option_2 : .....;  
..... : .....;  
default : .....;  
endcase
```

Control Statements: Loops

```
reg [3:0] i, output;  
for (i = 0; i <= 15; i = i + 1) begin  
    output = i;    #10;  
end
```

```
    reg [3:0] i, output;  
    i = 0;  
    while (i <= 15) begin  
        output = i;  
        #10 i = i + 1;  
    end
```

```
repeat (Times) begin  
    $display ("Current value of i is %d", i);  
    i = i + 1;  
end
```

Blocking vs. Nonblocking

- Verilog has two types of procedural assignment: Blocking and Nonblocking
- Fundamental problem:
 - In a synchronous system, all flip-flops sample simultaneously
 - In Verilog, *always @()* blocks run in some undefined sequence

A Flawed Shift Register

- This doesn't work as you'd expect:

```
reg d1, d2, d3, d4;  
always @(posedge clk) d2 = d1;  
always @(posedge clk) d3 = d2;  
always @(posedge clk) d4 = d3;
```

- These run in some order, but you don't know which

Non-blocking Assignments

- This version does work:

```
reg d1, d2, d3, d4;  
always @(posedge clk) d2 <= d1;  
always @(posedge clk) d3 <= d2;  
always @(posedge clk) d4 <= d3;
```

- Nonblocking rule: RHS evaluated when assignment runs
- LHS updated only after all events for the current instant have run

Nonblocking Can Behave Oddly

- A sequence of nonblocking assignments don't communicate

a = 1;

b = a;

c = b;

Blocking assignment:

a = b = c = 1

a <= 1;

b <= a;

c <= b;

Nonblocking assignment:

a = 1

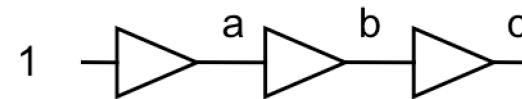
b = old value of a

c = old value of b

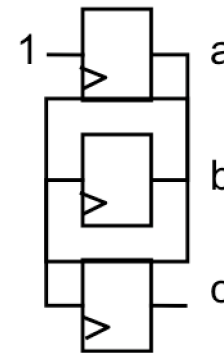
Nonblocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

`a = 1;`
`b = a;`
`c = b;`



`a <= 1;`
`b <= a;`
`c <= b;`

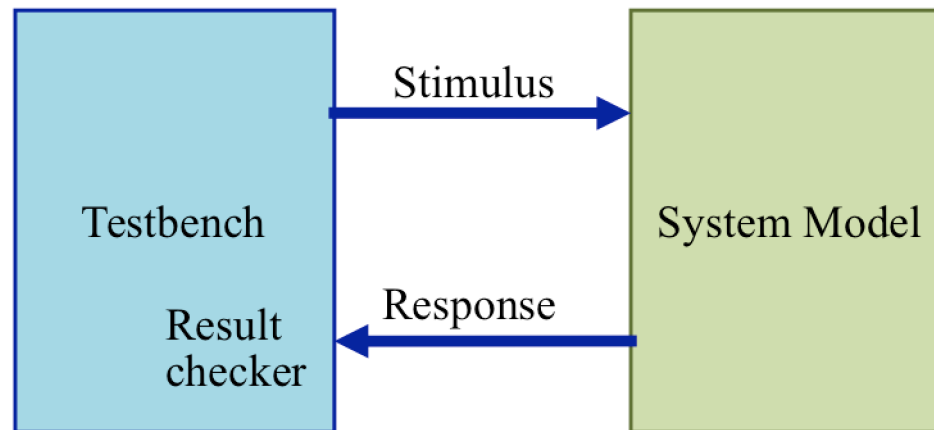


HDL Simulation

Simulation of Verilog Designs

How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously



Writing Testbenches

```
module test;
reg a, b, sel; //Inputs to device under test
mux m(y, a, b, sel); //Device under test

initial begin
    //$monitor is a built-in event driven “printf”
    $monitor($time, “a = %b b = %b sel = %b y
               = %b”, a, b, sel, y);
    //Stimulus generated by sequence of assignments
    //and delays
    a = 0; b = 0; sel = 0;
    #10 a = 1;
    #10 sel = 1;
    #10 b = 1;
end
```

Simulation Behavior

- Scheduled using an event queue
- Non-preemptive, no priorities
- A process must explicitly request a context switch
- Events at a particular time unordered
- Scheduler runs each event at the current time, possibly scheduling more as a result

Two Types of Events

- Evaluation events compute functions of inputs
- Update events change outputs
- Split necessary for delays, nonblocking assignments, etc.

Update event writes
new value of **a** and
schedules any
evaluation events
that are sensitive to
a change on **a**

$a \leq b + c$

Evaluation event reads
values of **b** and **c**, adds
them, and schedules an
update event

Simulation Behavior

- Concurrent processes (initial, always) run until they stop at one of the following
 - #42
 - Schedule process to resume 42 time units from now
 - **wait**(X & Y)
 - Resume when expression “X & Y” becomes true
 - (a or b or y)
 - Resume when a, b, or y changes
 - (posedge clk)
 - Resume when **clk** changes from 0 to 1

Simulation Behavior

- Infinite loops are possible and the simulator does not check for them
- This runs forever: no context switch allowed, so ready can never change

```
while (~ready)
    count = count + 1;
```

- Instead, use

```
wait(ready);
```

Simulation Behavior

- Race conditions abound in Verilog
- These can execute in either order: final value of a undefined:

```
always @(posedge clk) a = 0;
```

```
always @(posedge clk) a = 1;
```

Simulation Behavior

- Semantics of the language closely tied to simulator implementation
- Context switching behavior convenient for simulation, not always best way to model
- Undefined execution order convenient for implementing event queue

References

- 1) Morris Mano, “Digital Design: with an Introduction to the Verilog HDL”, Pearson Education, 5 Edition, 2014
- 2) David Money Harris and Sarah L. Harris, “Digital Design and Computer Architecture”, Elsevier, 2007
- 3) Stephen A. Edwards, “The Verilog Language”, CS dept., Columbia University, 2001
- 4) Samir Palnitkar, “Verilog HDL”, Pearson Education, 2 Edition, 2003