Learning Objectives

- Acquire a basic knowledge of the Verilog HDL
  - Syntax and lexical conventions
  - Data types, operators, expressions, and assignments
  - Structural primitives
  - Structural and behavioral models
  - Testbenches for simulation and verification
- Read and write simple Verilog models of basic constructs of the Verilog HDL
- Synthesize FPGAs from HDL models
- Learn a methodology for designing, verifying, and synthesizing a FSM controller for a datapath in a digital system
- Write race-free, latch-free synthesizable models

HDL Styles of Models

Example: Compare two 2-bit binary words:

A.lt.B = A' B1 + A1' A0' B0 + A0' B1 B0
A.gt.B = A1 B1' + A0 B1' B0' + A1 A0 B0'
A.eq.B = A1' A0' B1' B0' + A1' A0 B1' B0 + A1 A0 B1 B0 + A1 A0' B1 B0'

- Classical approach: write logic equations, reduce, and produce the schematic
- Structural HDL approach: Connect primitives to describe the functionality implied by the schematic
- Behavioral HDL approach: Write an RTL/algorithm description of the functionality, then synthesize a physical implementation

HDL Example: Half Adder - Structural Model

Verilog primitives encapsulate pre-defined functionality of common logic gates.

- The counterpart of a schematic is a structural model composed of Verilog primitives
- The model describes relationships between outputs and inputs

```verilog
module Add_half (sum, c_out, a, b);
  input [1:0] a, b;
  output [1:0] sum, c_out;
  xor (sum, a, b);
  and (c_out, a, b);
endmodule
```

```verilog
module Add_half (output sum, c_out, input a, b); // ANSI C style
  xor (sum, a, b);
  and (c_out, a, b);
endmodule
```
Primitives

Built-in primitives (combinational):

<table>
<thead>
<tr>
<th>n-Input</th>
<th>n-Output, 3-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>buf</td>
</tr>
<tr>
<td>nand</td>
<td>not</td>
</tr>
<tr>
<td>or</td>
<td>bufif0</td>
</tr>
<tr>
<td>nor</td>
<td>bufif1</td>
</tr>
<tr>
<td>xor</td>
<td>notif0</td>
</tr>
<tr>
<td>xnor</td>
<td>notif0</td>
</tr>
</tbody>
</table>

Structural Connectivity

- Wires in Verilog establish connectivity between primitives and/or modules
- Data type: nets (Example: wire)
- The logic value of a wire (net) is determined dynamically during simulation by what is connected to the wire.

Modeling Tip

Use wires to establish structural connectivity. An undeclared identifier is treated as a wire.

Modeling with Primitives

Model structural detail by instantiating and connecting primitives.

Example:

```
wire y1, y2;
nor (y_out, y1, y2);
and (y1, x_in1, x_in2);
nand (y2, x_in3, x_in4, x_in5);
```

Design Encapsulation

Encapsulate structural and functional details in a module.

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

Encapsulation makes the model available for instantiation in other modules.

Modeling Tip

The output port of a primitive must be first in the list of ports.
**Language Rules**

- Verilog is a case sensitive language (with a few exceptions)
- 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)
  - Max length of 1024 symbols
- Terminate lines with semicolon (;)
- Single line comments: // A single-line comment goes here
- Multi-line comments:

  /* Do not */ nest multi-line comments*/ like this */

**Representation of Numbers**

- Sized numbers specify the number of bits that are to be stored for a value
- Base specifiers:
  - b or B binary
  - d or D decimal (default)
  - o or O octal
  - h or H hexadecimal
- Examples:
  - 8'b1001_1101 // Use underscore for readability
  - 32'HABCD // Pads 0s
- Note: Unsized numbers are stored as integers (at least 32 bits)

**Structural Decomposition – Top-Down Design (1 of 2)**

Model complex structural detail by instantiating modules within modules

**Structural Decomposition – Top-Down Design (2 of 2)**

Modeling Tip:
The ports of a module may be listed in any order.
The instance name of a primitive is optional.
Port Connection by Name

Connect ports by name in modules that have several ports.

```
Add_half_0_delay M1 ( .b (b),
  .c_out (w2),
  .a (a),
  .sum (w1)
);
```

Logic System

- Four values: 0, 1, x or X, z or Z  // Not case sensitive here
- Primitives have built-in 4-value logic
- Simulators describe 4-value logic

Modeling Tip:

Logic value x denotes an unknown (ambiguous) value.
Logic value z denotes a high impedance.

Example: 4-Valued Logic Gate

Resolution of Contention

The value on a wire having multiple drivers in contention may be x.
Preview: Behavioral Modeling with Verilog

- Three types of behaviors for composing abstract models
  - Continuous assignment (Keyword: assign) – Boolean logic
  - Single pass behavior (Keyword: initial) – Use only in testbenches
  - Cyclic behavior (Keyword: always) – Level-sensitive (combinational logic) and edge-sensitive (synchronous logic)

- Single pass and cyclic behaviors execute procedural statements like those executed by a programming language (case, if, for, repeat …)

- Procedural assignment statements execute sequentially ( = ) or concurrently ( <= ), depending on the assignment operator

- A single pass behavior expires after the last statement executes

- A cyclic behavior executes again, repeatedly, after the last statement executes

---

Example: Testbench for Half Adder

```verilog
module t_Add_half ( );
wire sum, c_out;
reg a, b; // Storage containers for stimulus waveforms
Add_half_0_delay M1 (sum, c_out, a, b); // UUT

initial begin
#100 $finish; end // Optional stopwatch

initial begin
// Statements execute in sequence
#10 a = 0; b = 0; // Execution delays accumulate
#10 b = 1;
#10 a = 1;
#10 b = 0; // Executes at tsim = 40
end
endmodule
```

---

Template for Testbenches

```verilog
module t_DUTB_name ( ); // Substitute the name of the UUT
reg …; // Declare regs for inputs of the UUT
wire …; // Declare wires for outputs of the UUT
parameter time_out = 1; // Provide a value
UUT_name M1_instance_name ( UUT ports go here);
initial $monitor ( ); // Specify signals to be displayed as text
initial #time_out $finish; // (Also $stop) Stopwatch
initial // Stimulus patterns
begin // ...
// Behavioral statements
end
endmodule
```

---

Example: Clock Generator

```verilog
…
parameter latency = 50, half_cycle = 10;
…
initial #latency forever # half_cycle clock = ~ clock;
…
```

---
Continuous Assignments

- Continuous assignments (Keyword: `assign`) are the Verilog counterpart of Boolean equations.
- Hardware is implicit (e.g., combinational logic).

Example:

```verilog
module my_nand (
    output y,
    input x1, x2
);
    assign y = ~(x1 & x2);
endmodule
```

The RHS expression is monitored, and the LHS variable is updated automatically when the RHS expression changes value.

Example: And-Or-Invert

```verilog
module AOI_5_CA1 (output y_out, input x_in1, x_in2, x_in3, x_in4, x_in5);
    assign y_out = ~( (x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));
endmodule
```

---

Example: 2-Bit Comparator

Using primitives

```verilog
module compare_2_str (A_gt_B, A_lt_B, A_eq_B, A0, A1, B0, B1);
    output A_gt_B, A_lt_B, A_eq_B;
    input A0, A1, B0, B1;
    // Note: w1, w2, ... are implicit wires
    nor (A_gt_B, A_lt_B, A_eq_B);
    or (A_gt_B, w1, w2, w3);
    and (A_eq_B, w4, w5);
    and (w1, w6, B1);
    and (w2, w6, w7, B0);
    and (w3, w7, B0, B1); // Note: interchanging w7, B0 and B1 has no effect
    not (w6, A1);
    not (w7, A0);
    xnor (w4, A1, B1);
    xnor (w5, A0, B0);
endmodule
```

Note: n inputs are accommodated automatically.
**Concurrent Continuous Assignments**

Multiple continuous assignments are active concurrently.

**Example: 2-Bit Comparator**

```verilog
module compare_2_CA0 (  
    output A_lt_B, A_gt_B, A_eq_B, input A1, A0, B1, B0  
);
  assign A_lt_B = (~A1) & B1 | (~A1) & (~A0) & B0 | (~A0) & B1 & B0;
  assign A_gt_B = A1 & (~B1) | A0 & (~B1) & (~B0) | A1 & A0 & (~B0);
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) | (~A1) & A0 & (~B1)  
    & B0 | A1 & A0 & B1 & B0 | A1 & (~A0) & B1 & (~B0);
endmodule
```

**Alternative Models: Concatenation**

```verilog
module compare_2_CA1 (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);  
  input A1, A0, B1, B0;  
  output A_lt_B, A_gt_B, A_eq_B;  
  assign A_lt_B = ({A1, A0} < {B1, B0});  
  assign A_gt_B = ({A1, A0} > {B1, B0});  
  assign A_eq_B = ({A1, A0} == {B1, B0});
endmodule
```

**Alternative Models: Busses**

```verilog
module compare_2_CA1 (A_lt_B, A_gt_B, A_eq_B, A, B);  
  input [1: 0] A, B;  
  output A_lt_B, A_gt_B, A_eq_B;  
  assign A_lt_B = (A < B);  // RHS is true (1) or false (0)  
  assign A_gt_B = (A > B);  
  assign A_eq_B = (A == B);
endmodule
```

**Modeling Combinational Logic With Cyclic Behaviors**

**Example: 32-Bit Comparator**

```verilog
module compare_32_CA (  
    input [31: 0] A, B, output reg A_gt_B, A_lt_B, A_eq_B  
);
  always @ (A, B) begin  
    A_gt_B <= (A > B), // List of multiple procedural assignments  
    A_lt_B <= (A < B),  
    A_eq_B <= (A == B);
  end
endmodule
```

The target of a procedural assignment in a cyclic behavior must be a declared `reg`. 
Truth Table Using Case Statement

module divisible_by_3(A, F);
input [2:0] A;
output F;
reg F;
always @ (A)
    case (A)
        0: F <= 0;
        1: F <= 0;
        2: F <= 0;
        3: F <= 1;
        4: F <= 0;
        5: F <= 0;
        6: F <= 1;
        7: F <= 0;
    endcase
endmodule

Modeling Synchronous Logic with Cyclic Behaviors

- Use edge-sensitive cyclic behaviors to model flip-flops and sequential logic.

 always @ (posedge clock, negedge reset)
 begin
     if (reset == 0) q <= 0;
     else if (set == 0) q <= 1;
     else q <= data;
 end

How does a synthesis tool identify the synchronizing signal?

Modeling Synchronous Logic (1 of 2)

Example: D Flip-Flop with asynchronous set / reset (active low)

module df_behav (input data, set, clk, reset, output reg q, output q_bar);
    assign q_bar = ~ q;
    always @ (posedge clk, negedge reset, negedge set)
    begin
        if (reset == 0) q <= 0;
        else if (set == 0) q <= 1;
        else q <= data;
    end
endmodule

Non-blocking assignment operator

Modeling Synchronous Logic (2 of 2)

Example: D flip-flop with synchronous set / reset

module df_behav (input data, set, clk, reset, output reg q, output q_bar);
    assign q_bar = ~ q;
    always @ (posedge clk)
    begin
        if (reset == 0) q <= 0;
        else if (set == 0) q <= 1;
        else q <= data;
    end
endmodule
Dataflow / RTL Behavioral Modeling

- Dataflow / RTL (register transfer level) models of combinational logic describe concurrent operations on datapath signals, usually in a synchronous machine.
- Register operations with language operators
- Assignment of value to variables
- Control flow constructs

Verilog Operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>Logic Not</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>Bitwise Not</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>Left Shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>Right Shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;</td>
<td>Logical Left Shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;</td>
<td>Logical Right Shift</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td></td>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Not Equal</td>
</tr>
<tr>
<td></td>
<td>===</td>
<td>Strict Equality</td>
</tr>
<tr>
<td></td>
<td>!==</td>
<td>Strict Not Equal</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>XOR</td>
</tr>
<tr>
<td></td>
<td>^~</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td>~^</td>
<td>Bitwise NOT XOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td></td>
<td>? :</td>
<td>Conditional</td>
</tr>
<tr>
<td>Lowest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adder The Easy Way!

- Continuous assignments (Keyword: assign) are the Verilog counterpart of Boolean equations
- Hardware is implicit (e.g., combinational logic)

```verilog
module add4 (A, B, S);
  input [3:0] A;
  input [3:0] B;
  output [4:0] S;
  assign S = A + B;
endmodule
```

RTL Example: shift Register

```verilog
module Shift_reg4 (output Data_out, input Data_in, clock, reset);
  reg [3:0] Data_reg;
  assign Data_reg <= {Data_in, Data_reg[3:1]};
  always @ (negedge reset or posedge clock)
    begin
      if (reset == 1'b0) Data_reg <= 4'b0;
      else              Data_reg <= {Data_in, Data_reg[3:1]};
    end
endmodule
```
Concurrent Assignments (Non-Blocking)

- Nonblocking assignment \( (<=) \) statements execute concurrently (in parallel) rather than sequentially.
- The order in which nonblocking assignments are listed has no effect.
- Mechanism: the RHS of the assignments are sampled, then assignments are updated.
- Assignments are based on values held by RHS before the statements execute.
- Result: No dependency between statements.

Example: Concurrent Assignment

```verilog
module shiftreg_PA (E, A, clk, rst);
    output A;
    input E;
    input clk, rst;
    reg A, B, C, D;

    always @ (posedge clk or posedge rst) begin
        if (reset) begin
            A = 0; B = 0; C = 0; D = 0;
        end else begin
            A <= B;
            B <= C;
            C <= D;
            D <= E;
        end
    end
endmodule
```

Sequential Assignment

- The order of execution of procedural statements in a cyclic behavior may depend on the order in which the statements are listed.
- A blocked procedural assignment \( (=) \) cannot execute until the previous statement executes.
- We're not going to use this in this class!