Lecture 2: Data Types, Modeling
Combinational Logic in Verilog HDL

Why use an HDL?
Increase digital design engineer’s productivity
(from Dataquest)
Behavioral HDL 2K – 10K gates/week
RTL HDL 1K – 2K gates/week
Gates 100 – 200 gates/week
Transistors 10 – 20 gates/week

Variables and Logic Value Set

Variables: represent the values of signals in a circuit
Two kinds of variables: Nets and Registers
   Nets: represent the structural connectivity in a circuit
   Registers: represent storage elements

Logic Value Set
<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic 0, or false condition</td>
</tr>
<tr>
<td>1</td>
<td>Logic 1, or true condition</td>
</tr>
<tr>
<td>x</td>
<td>represent an unknown logic value</td>
</tr>
<tr>
<td>z</td>
<td>represent a high impedance condition</td>
</tr>
</tbody>
</table>

Data Types

- Nets
  - Nets are physical connections between devices
  - Nets always reflect the logic value of the driving device
  - Many types of nets, but all we care about is wire

- Registers
  - Implicit storage – unless variable of this type is modified it retains previously assigned value
  - Does not necessarily imply a hardware register
  - Register type is denoted by reg
  - int is also used
**Data Types: Nets**

**Nets for connectivity:**
- **wire** establishes connectivity
- **tri** same as wire and it will be tri-stated in hardware
- **wand** a net has multiple drivers, wires and, i.e., open collector circuit
- **wor** a net has multiple drivers, wired or, i.e., emitter coupled circuit
- **triand** a net that has multiple drivers. It models wired-and. It is tri-stated.
- **tior** a net that has multiple drivers. It models wired-or. It is tri-stated.
- **supply0** a global net connected to the circuit ground
- **supply1** a global net connected to the power supply
- **tri0** a net connected to the ground by a resistive pulldown connection.
- **tri1** a net connected to the power supply by a resistive pullup connection.
- **trireg** a net that models the charge stored on a physical net.

---

**Data Types: Registers**

**Registers for storage**
A register variable is an abstraction of a hardware storage element.

**Rising Edge Flip-Flop**

```
module dff (data, clk, q);
input data, clk;
output q;
reg q;
always @ (posedge clk)
q <= data; // Tools require left-hand side must be a register
// for statements in an always block
endmodule
```

*"<="* is non-blocking assignment for flip flops
More about "<=" non-blocking, "=" block assignment later

---

**Rising Edge Flip-Flop with Asynchronous Reset**

```
module dff_async_rst (data, clk, reset, q);
input data, clk, reset;
output q;
reg q;
always @(posedge clk or negedge reset)
if (~reset)
q <= 1'b0;
else
q <= data;
endmodule
```

---

**Variables referenced, but undeclared are implicit wires**

Gates drive nets. The output of a gate by default is a wire

```
module Add_half (sum, c_out, a, b);
output sum, c_out;  // declare output port of type net, actually wire
input a, b;          // declare input port of type net, actually wire
wire c_out_bar;
xor G1 (sum, a, b);
nand G2 (c_out_bar, a, b);
not G3 (c_out, c_out_bar);
endmodule
```

These two modules are equivalent

<table>
<thead>
<tr>
<th>input port of type net</th>
<th>output port of type net</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
Rising Edge Flip-Flop with Asynchronous Preset

```verilog
module dff_async_pre (data, clk, preset, q);
  input data, clk, preset;
  output q;
  reg q;
  always @(posedge clk or negedge preset)
    if (~preset)
      q <= 1'b1;
    else
      q <= data;
endmodule
```

Rising Edge Flip-Flop with Synchronous Reset

```verilog
module dff_sync_rst (data, clk, reset, q);
  input data, clk, reset;
  output q;
  reg q;
  always @(posedge clk)
    if (~reset)
      q <= 1'b0;
    else
      q <= data;
endmodule
```

Rising Edge Flip-Flop with Asynchronous Reset and Preset

```verilog
module dff_async (reset, preset, data, q, clk);
  input clk;
  input reset, preset, data;
  output q;
  reg q;
  always @(posedge clk or negedge reset or posedge preset)
    if (~reset)
      q <= 1'b0;
    else if (preset)
      q <= 1'b1;
    else
      q <= data;
endmodule
```

Rising Edge Flip-Flop with Synchronous Preset

```verilog
module dff_sync_pre (data, clk, preset, q);
  input data, clk, preset;
  output q;
  reg q;
  always @(posedge clk)
    if (~preset)
      q <= 1'b1;
    else
      q <= data;
endmodule
```
D-Latch with Data and Enable

module d_latch (enable, data, y);
input enable, data;
output y;
reg y;
always @ (enable or data)
if (enable)
y <= data;
endmodule

D-Latch with Gated Asynchronous Data

module d_latch_e(enable, gate, data, q);
input enable, gate, data;
output q;
reg q;
always @ (enable or data or gate)
if (enable)
q <= (data & gate);
endmodule

D-Latch with Gated Enable

module d_latch_en(enable, gate, d, q);
input enable, gate, d;
output q;
reg q;
always @ (enable or d or gate)
if (enable & gate)
q <= d;
endmodule

Net Declaration

wire [7:0] data_bus; // 8 bit bus, data_bus[7] is MSB
wire [0:3] control_bus; // control_bus[0] is MSB

//access bus examples
data_bus[3] // access data_bus bit 3
data_bus[3:0] // access bit 3 to bit 0 of data_bus
data_bus[k+2] // access a bit of the data_bus,
// depending on k+2

wire y, x, z; // y, x, z are three wires
wand A, B, C; // A, B, C wired and nets

Undeclared nets will default implicitly to type wire.
What if a wire or tri type net is driven by multiple drivers?

Verilog issues a warning and determines the value by pairwise application of the following table

<table>
<thead>
<tr>
<th>wire/tri</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

Design engineers don't want to drive a wire with more than one signal.

What is the initial value of a net?

A net driven by a primitive, module, or continuous assignment has a value "x" at the start of simulation. A net without any drivers is default to "z".

wire a, b, c;
assign a = b+ c; // initial value by default b = z, c = z, a = x

The initial value for a register variable is by default also "x".

Register Data Types

Register Data Types: reg, integer, time, realtime

<table>
<thead>
<tr>
<th>Register type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>Stores a logic value</td>
</tr>
<tr>
<td>integer</td>
<td>Supports computation</td>
</tr>
<tr>
<td>time</td>
<td>Supports time as a 64-bit unsigned number</td>
</tr>
<tr>
<td>real</td>
<td>Stores values (e.g., delay) as real numbers</td>
</tr>
<tr>
<td>realtime</td>
<td>Stores time values as real numbers</td>
</tr>
</tbody>
</table>

A register may never be the output of a primitive gate, or the target of a continuous assignment

(module adder_4_RTL (a, b, c_in, sum, c_out);
  output [3:0] sum;
  output c_out;
  input [3:0] a, b;
  input c_in;

  assign {c_out, sum} = a + b + c_in;
  // continuous assignment, any change of a, b, c_in
  // Verilog re-evaluates the output
  endmodule)

Continuous assignment allows you to specify combinational logic in equation form. Anytime an input (value on the right-hand side) changes, the simulator re-evaluates the output

No gate structure is implied — logic synthesis can design it.)
Verilog has the following operators for continuous assignments and register manipulations:

**Arithmetic Operators:** +, -, *, /, % (modulus)

**Bitwise/Logical Operators**
- Bitwise operators operate on the bits of the operand or operands.
  - For example, the result of A & B is the AND of each corresponding bit of A with B.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^ or ^~</td>
<td>Equivalence (Bitwise NOT XOR)</td>
</tr>
</tbody>
</table>

**{, }** concatenation

**Reduction Operators**
Reduction Operators: producing a single bit value by operating on a single data word.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reduction and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction nand</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>reduction exclusive or</td>
</tr>
<tr>
<td>^</td>
<td>reduction xnor</td>
</tr>
</tbody>
</table>

**Verilog Example**

```verilog
module synTriState (bus, in, driveEnable);
  input in, driveEnable;
  output bus;
  reg bus;
  always @(in or driveEnable)
  begin
    if (driveEnable)
      bus = in;
    else
      bus = 1`bz;
  end
endmodule
```

Wait for any change on in, driveEnable then execute the begin-end block containing the if. Then wait for another change.
### Initial Value of a Register Variable

```vhdl
reg A, B;
initial
begin
    A = 0;  // assign an initial value to A
    B = 1;  // assign an initial value to B
end
// All registers have an initial value of "x" by default.
```

### Passing Variables Through Ports

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Port Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>net variable</td>
<td>yes</td>
</tr>
<tr>
<td>register variable</td>
<td>no</td>
</tr>
</tbody>
</table>

- input port of a module is of type net.
- output port of a module is of type net, or reg.
- inout port of a module is of type net.

### Memory Declaration

```vhdl
reg [31:0] m [0:8191];   // 8192 x 32 bit memory
reg [15:0] pc;           // 16 bit program counter
reg [31:0] acc;          // 32 bit accumulator
reg [15:0] ir;           // 16 bit instruction register
reg ck;                  // a clock signal
```

### Hierarchical De-referencing

```vhdl
module test_Add_rca_4();
reg [3:0] a,b;
reg c_in;
wire [3:0] sum;
wire c_out;
initial
begin
    $monitor ($time,, "c_out= %b  c_in4=%b  c_in3=%b  c_in2=%b  c_in=%b ", c_out, M1.c_in4, M1.c_in3, M1.c_in2, c_in);
end
initial
begin
    // stimulus patterns generated here
end
Add_rca_4 M1 (sum, c_out, a, b, c_in);
// Add_rca_4 in next slide
endmodule
```
Verilog model: 4 bit RCA

module Add_rca_4 (sum, c_out, a, b, c_in);
  output [3:0] sum;
  output c_out;
  input [3:0] a, b;
  input c_in;
  wire c_out, c_in4, c_in3, c_in2;
  Add_full G1 (sum[0], c_in2, a[0], b[0], c_in);
  Add_full G2 (sum[1], c_in3, a[1], b[1], c_in2);
  Add_full G3 (sum[2], c_in4, a[2], b[2], c_in3);
  Add_full G2 (sum[3], c_out, a[3], b[3], c_in4);
endmodule

module Add_full(sum, cOut, aIn, bIn, cIn);
  output sum, cOut;
  input aIn, bIn, cIn;
  nand (x2, aIn, bIn),
  xnor (x9, x5, x6);
  nor (x5, x1, x3),
  or (x8, x1, x7);
  not (sum, x9),
  (x3, x2),
  (x6, x4),
  (x4, cIn),
  (x7, x6);
endmodule

A full adder uses Verilog primitive logic gates. Anytime the input to a gate changes, its output is evaluated, via its output wire to other inputs.

A default gate delay is 0

Parameters Substitution

module modXnor (y_out, a, b);
  parameter size=8, delay=15;
  output [size-1:0] y_out;
  input [size-1:0] a, b;
  wire [size-1:0] #delay y_out=a~^b;
endmodule

module modXnor (y_out, a, b);
  parameter size=8, delay=15;
  output [size-1:0] y_out;
  input [size-1:0] a, b;
  wire [size-1:0] #delay y_out=a~^b;
endmodule

module Param;
  wire [7:0] y1_out;
  wire [3:0] y2_out;
  reg [7:0] b1, c1;
  reg [3:0] b2, c2;
  modXnor G1 (y1_out, b1, c1);
  modXnor #(4, 5) G2 (y2_out, b2, c2); // size = 4, delay =5
endmodule

Indirect Parameters Substitution

module modXnor (y_out, a, b);
  parameter size=8, delay=15;
  output [size-1:0] y_out;
  input [size-1:0] a, b;
  wire [size-1:0] #delay y_out=a~^b;
endmodule

module modXnor (y_out, a, b);
  parameter size=8, delay=15;
  output [size-1:0] y_out;
  input [size-1:0] a, b;
  wire [size-1:0] #delay y_out=a~^b;
endmodule

module hdref_Param;
  wire [7:0] y1_out;
  wire [3:0] y2_out;
  reg [7:0] b1, c1;
  reg [3:0] b2, c2;
  modXnor G1 (y1_out, b1, c1);
  modXnor G2 (y2_out, b2, c2);
endmodule

module annotate;
  defparam hdref_Param.G2.size = 4,
  hdref_Param.G2.delay = 5;
endmodule
Verilog Model Example

Design a 4-to-1 mux by cascading 2-to-1 muxes.

module mux2to1 (f, a, b, sel);
output f;
input a, b, sel;
and g1 (f1, a, nsel),
g2 (f2, b, sel);
or g3 (f, f1, f2);
not g4 (nsel, sel);
endmodule

module mux4to1 (f, a, b, c, d, sel0, sel1);
output f;
input a, b, c, d, sel0, sel1;
wire w1, w2;
mux2to1 m1 (w1, a, b, sel0),
m2 (w2, c, d, sel0),
m3 (f, w1, w2, sel1);
endmodule

module test_mux4to1 (a, b, c, d, sel0, sel1, f);
// generating all inputs to the mux4to1, 
// receiving f from the mux4to1 output
input f;
output a, b, c, d, sel0, sel1;
reg a, b, c, d, sel0, sel1;
initial begin
$monitor ($time,, "a =%b, b =%b, c =%b, d =%b, sel1 =%b, sel0 =%b, f =%b", a, b, c, d, sel1, sel0, f);
a = 1; b =0; c =1; d =0; sel1 = 0; sel0 = 0;
#10 sel1= 0; sel0 = 1;
#10 sel1= 1; sel0 = 0;
#10 sel1= 1; sel0 = 1;
#10 a = 0; b =0; c = 1; d = 1;
#10 $finish;
end
endmodule

module testbench;
wire a, b, c, d, sel0, sel1, f;
test_mux4to1 my_tester (a, b, c, d, sel0, sel1, f);
mux4to1 my_design (f, a, b, c, d, sel0, sel1);
endmodule

copied from Silos output window

Ready: sim
10 a = 1, b = 0, c = 1, d = 0, sel1 = 0, sel0 = 1, f = 0
20 a = 1, b = 0, c = 1, d = 0, sel1 = 1, sel0 = 0, f = 1
30 a = 1, b = 0, c = 1, d = 0, sel1 = 1, sel0 = 1, f = 0
40 a = 0, b = 0, c = 1, d = 1, sel1 = 0, sel0 = 0, f = 0
50 a = 0, b = 0, c = 1, d = 1, sel1 = 0, sel0 = 1, f = 0
60 a = 0, b = 0, c = 1, d = 1, sel1 = 1, sel0 = 0, f = 1
70 a = 0, b = 0, c = 1, d = 1, sel1 = 1, sel0 = 1, f = 1
75 State changes on observable nets.
Simulation stopped at the end of time 80.
A Behavioral Model for MUX

module mux (f, sel, b, c);
output f;
input sel, b, c;
reg f;

always @ (sel or b or c)
  if (sel == 1)
    f = b;
  else
    f = c;
endmodule

Behavioral model uses Always, Initial construct.

Behavioral Model for Combinational Logic

Each “always” statement turns into Boolean functions

module example (f, a, b, c);
output f;
input a, b, c;
reg f;

always @ (a or b or c)
begin
  logic...
end
endmodule

In Verilog, always statement is considered as procedural statement. Initial is another procedural statement.

The rules for specifying combinational logic using procedural statements

1. Every element of the input set must be in the sensitivity list
2. The combinational output must be assigned in every control path

module test_mux4to1 (a, b, c, d, sel0, sel1, f);
// generating all inputs to the mux4to1, receiving f from the mux4to1 output
input f;
output a, b, c, d, sel0, sel1;
reg a, b, c, d, sel0, sel1;
initial begin
$monitor ($time,, "a = %b, b = %b, c = %b, d = %b, sel1 = %b, sel0 = %b, f = %b", a, b, c, d, sel1, sel0, f);
  a = 1; b = 0; c = 1; d = 0; sel1 = 0; sel0 = 0;
  sel1 = 1; sel0 = 0;
  #10 sel1 = 1; sel0 = 1;
  #10 a = 0; b = 0; c = 1; d = 1; sel1 = 0; sel0 = 0;
  #10 sel1 = 1; sel0 = 1;
  #10 sel1 = 1; sel0 = 0;
  #10 sel1 = 1; sel0 = 1;
  $finish;
end
endmodule

module exam1 (f, sel, b, c);
output f; input sel, b, c;
reg f;
always @ (sel or b or c)
if (sel == 1)
  f = b;
else
  f = c;
endmodule

module exam2 (f, g, sel, b, c);
output f, g; input sel, b, c;
reg f, g;
always @ (sel or b or c)
if (sel == 1)
  f = b;
else
  g = c;
endmodule

Wrong model
Behavioral Model for Combinational Logic: Case statement

Generate a truth table, assign output \( f \) in each case item

```verilog
module logic1 (f, a, b, c);
output f;
input a, b, c;
reg f;
always @ (a or b or c)
case ({a, b, c})
  3'b000: f = 1'b0;
  3'b001: f = 1'b1;
  3'b010: f = 1'b1;
  3'b011: f = 1'b1;
  3'b100: f = 1'b1;
  3'b101: f = 1'b0;
  3'b110: f = 1'b0;
  3'b111: f = 1'b1;
endcase
endmodule
```

Verilog behavioral model for combinational logic

```verilog
module name (<output names>, <input names>);
output <output names>;
input <input names>;
reg <output names>;
always @ (<names of all input vars>)
begin
  < LHS = RHS assignments>
  < if ... else statements>
  < case statements >
end
endmodule
```