# EECS150 - Digital Design
## Lecture 5 - Verilog 2

**February 1, 2005**

John Wawrzynek

## Outline

- Verilog Basics
- Lots of Examples

---

### Structural Model - XOR

```verilog
module xor_gate (out, a, b);
    input a, b;
    output out;
    wire abar, bbar, t1, t2;

    not invA (abar, a);
    not invB (bbar, b);
    and and1 (t1, a, bbar);
    and and2 (t2, b, abar);
    or or1 (out, t1, t2);

endmodule
```

- **Structural Model - XOR**
  - Composition of primitive gates to form more complex module
  - Note use of *wire* declaration!

### Structural Model: 2-to1 mux

```verilog
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0, w1;

    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or  (out, w0, w1);

endmodule
```

- **Notes:**
  - comments
  - "module"
  - port list
  - declarations
  - wire type
  - primitive gates
  - Instance names?
  - List per type
Simple Behavioral Model

- Combinational logic Example
  - Describe output as a function of inputs
  - Note use of `assign` keyword: *continuous* assignment

```verilog
module and_gate (out, in1, in2);
    input in1, in2;
    output out;
    assign out = in1 & in2;
endmodule
```

Verilog Continuous Assignment

- Assignment is continuously evaluated
- `assign` corresponds to a connection or a simple component with the described function
- Target is NEVER a reg variable
  - “Dataflow” style

```verilog
assign A = X | (Y & ~Z);
assign B[3:0] = 4'b01XX;
assign C[15:0] = 4'h00ff;
```

Comparator Example

```verilog
module Compare1 (A, B, Equal, Alarger, Blarger);
    input A, B;
    output Equal, Alarger, Blarger;
    assign Equal = (A & B) | (~A & ~B);
    assign Alarger = (A & ~B);
    assign Blarger = (~A & B);
endmodule
```

Comparator Example

```verilog
// Make a 4-bit comparator from 4 1-bit comparators
module Compare4(A4, B4, Equal, Alarger, Blarger);
    input [3:0] A4, B4;
    output Equal, Alarger, Blarger;
    wire e0, e1, e2, e3, A0, A1, A2, A3, B0, B1, B2, B3;
    Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
    Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
    Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
    Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);
    assign Equal = (e0 & e1 & e2 & e3);
    assign Alarger = (Al3 | (Al2 & e3) | (Al1 & e3 & e2) | (A0 & e3 & e2 & e1));
    assign Blarger = (~Alarger & ~Equal);
endmodule
```
Simple Behavioral Model - the *always* block

- **always block**
  - Always waiting for a change to a trigger signal
  - Then executes the body

```verilog
module and_gate (out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @(in1 or in2) begin
    out = in1 & in2;
  end
endmodule
```

**Notes:**
- Behavioral descriptions use the keyword *always* followed by blocking procedural assignments.
- Target output of procedural assignments must of of type *reg* (not a real register).
- Unlike wire types where the target output of an assignment may be continuously updated, a *reg* type retains its value until a new value is assigned (the assigning statement is executed).
- Optional initial statement

**always Block**

- A procedure that describes the function of a circuit
  - Can contain many statements including if, for, while, case
  - Statements in the *always* block are executed sequentially
    - (Continuous assignments <= are executed in parallel)
  - The entire block is executed at one
  - The *final* result describes the function of the circuit for current set of inputs
  - Intermediate assignments don't matter, only the final result
- *begin/end* used to group statements

---

**2-to-1 mux behavioral description**

// Behavioral model of 2-to-1
// multiplexor.
module mux2 (in0,in1,select,out);
  input in0,in1,select;
  output out;
  reg out;
  //
  reg out;
  always @(in0 or in1 or select) if (select) out=in1;
  else out=in0;
endmodule // mux2

**Notes:**
- Behavioral descriptions use the keyword *always* followed by blocking procedural assignments.
- Target output of procedural assignments must of of type *reg* (not a real register).
- Unlike wire types where the target output of an assignment may be continuously updated, a *reg* type retains its value until a new value is assigned (the assigning statement is executed).
- Optional initial statement

**Behavioral 4-to1 mux**

//Does not assume that we have // defined a 2-input mux.
module mux4 (in0, in1, in2, in3, select, out);
  input in0,in1,in2,in3;
  input [1:0] select;
  output      out;
  reg out;
  always @ (in0 in1 in2 in3 select)
  case (select)
    2'b00: out=in0;
    2'b01: out=in1;
    2'b10: out=in2;
    2'b11: out=in3;
  endcase
endmodule // mux4

**Notes:**
- No instantiation
- Case construct equivalent to nested if constructs.
- **Definition:** A structural description is one where the function of the module is defined by the instantiation and interconnection of sub-modules.
- A behavioral description uses higher level language constructs and operators.
- Verilog allows modules to mix both behavioral constructs and sub-module instantiation.
Mixed Structural/Behavioral Model

- Example 4-bit ripple adder

```verilog
module adderCell (S, Cout, A, B, Cin);
    input A, B, Cin;
    output S, Cout;
    assign {Cout, S} = A + B + Cin;
endmodule
```

```verilog
module adder4 (S, Cout, A, B, Cin);
    input [3:0] A, B;
    input Cin;
    output [3:0] S;
    output Cout;
    wire C1, C2, C3;
    addrCell fc0 (S[0], C1, A[0], B[0], Cin);
    addrCell fc1 (S[1], C2, A[1], B[1], C1);
    addrCell fc2 (S[2], C3, A[2], B[2], C2);
    addrCell fc3 (S[3], Cout, A[3], B[3], C3);
endmodule
```

Verilog Data Types and Values

- Bits - value on a wire
  - 0, 1
  - X - don’t care/don’t know
  - Z - undriven, tri-state
- Vectors of bits
  - Treated as an unsigned integer value
  - e.g., A < 0 ??
  - Concatenating bits/vectors into a vector
    - e.g., sign extend
  - Style: Use a[7:0] = b[7:0] + c;
    Not: a = b + c; // need to look at declaration

Verilog Help

- The lecture notes only cover the very basics of Verilog and mostly the conceptual issues.
- Textbook has examples.
- The Bhasker book is a good tutorial. On reserve in the Engineering library (starting Friday).
- The complete language specification from the IEEE is available on the class website under “Refs/Links”

Verilog Numbers

- 14 - ordinary decimal number
- -14 - 2’s complement representation
- 12'b0000_0100_0110 - binary number with 12 bits (___ is ignored)
- 12'h046 - hexadecimal number with 12 bits
- Verilog values are unsigned
  - if A = 0110 (6) and B = 1010(-6)
    C = 10000 not 00000
    i.e., B is zero-padded, not sign-extended
Verilog Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>bit-select or port-select</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td>Anything</td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XNOR</td>
</tr>
<tr>
<td><code>-</code></td>
<td>vee (logical plus)</td>
<td>Truthy</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>/</code></td>
<td>divide</td>
<td>Arithmatic</td>
</tr>
<tr>
<td><code>%</code></td>
<td>modulo</td>
<td>Arithmatic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>unequal</td>
<td>Equality</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^</td>
<td></td>
<td>Bit-wise</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bit-wise XNOR</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit-wise</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>logical AND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>logical OR</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>conditional</td>
</tr>
</tbody>
</table>

Verilog Variables

- **wire**
  - Variable used simply to connect components together
  - Wire is a thing of type “nettype”, other examples of nettype are: supply0 (connection to logic 0) and supply1 (connection to logic 1)

- **reg**
  - Variable that saves a value as part of a behavioral description
  - Usually corresponds to a wire in the circuit
  - Is NOT necessarily a register in the circuit

- **usage:**
  - Don’t confuse reg assignments with the combinational continuous assign statement! (more soon)
  - Reg should only be used with always blocks (sequential logic, to be presented …)

“Complete” Assignments

- If an **always** block executes, and a variable is **not** assigned
  - Variable keeps its old value (think implicit state!)
  - **NOT** combinational logic ⇒ latch is inserted (implied memory)
  - This is usually **not** what you want: dangerous for the novice!
- Any variable assigned in an **always** block should be assigned for any (and every!) execution of the block

Incomplete Triggers

- Leaving out an input trigger usually results in a sequential circuit (circuit with state)
- Example: The output of this “and” gate depends on the input history

```verilog
module and_gate (out, in1, in2);
    input in1, in2;
    output out;
    reg out;
    always @(in1) begin
        out = in1 & in2;
    end
endmodule
```
Behavioral with Bit Vectors

//Behavioral model of 32-bitwide 2-to-1 multiplexor.
module mux32 (in0,in1,select,out);
    input [31:0] in0,in1;
    input select;
    output [31:0] out;
    //
    reg [31:0] out;
    always @ (in0 or in1 or select)
        if (select) out=in1;
        else out=in0;
endmodule // Mux

//Behavioral model of 32-bit adder.
module add32 (S,A,B);
    input [31:0] A,B;
    output [31:0] S;
    reg [31:0] S;
    //
    always @ (A or B)
        S = A + B;
endmodule // Add

Verilog if

• Same as C if statement

    // Simple 4-1 mux
    module mux4 (sel, A, B, C, D, Y);
        input [1:0] sel; // 2-bit control signal
        input A, B, C, D;
        output Y;
        reg Y; // target of assignment
        always @(sel or A or B or C or D)
            if (sel == 2’b00) Y = A;
            else if (sel == 2’b01) Y = B;
            else if (sel == 2’b10) Y = C;
            else if (sel == 2’b11) Y = D;
endmodule

Hierarchy & Bit Vectors

// Assuming we have already
// defined a 2-input mux (either
// structurally or behaviorally,
// named port assignment
//4-input mux built from 3 2-input muxes
module mux4 (in0, in1, in2, in3, select, out);
    input in0, in1, in2, in3;
    input [1:0] select;
    output out;
    wire w0, w1;
    m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
    m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
    m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4

Verilog if

    // Simple 4-1 mux
    module mux4 (sel, A, B, C, D, Y);
        input [1:0] sel; // 2-bit control signal
        input A, B, C, D;
        output Y;
        reg Y; // target of assignment
        always @(sel or A or B or C or D)
            if (sel[0] == 0)
                if (sel[1] == 0) Y = A;
                else         Y = B;
            else
                if (sel[1] == 0) Y = C;
                else         Y = D;
endmodule
Verilog case

- Sequential execution of cases
  - Only first case that matches is executed (no break)
  - Default case can be used

```verilog
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;
reg Y; // target of assignment
always @(sel or A or B or C or D)
  case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
  endcase
endmodule
```

Verilog case

- Without the default case, this example would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

```verilog
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
  case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:     Y = 3'bX; // Don’t care when input is not 1-hot
  endcase
endmodule
```

Verilog case (cont)

- Cases are executed sequentially
  - The following implements a priority encoder

```verilog
// Priority encoder
module encode (A, Y);
input  [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
  case (A)
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3'bX; // Don’t care when input is all 0’s
  endcase
endmodule
```

Parallel Case

- A priority encoder is more expensive than a simple encoder
  - If we know the input is 1-hot, we can tell the synthesis tools
  - "parallel-case" pragma says the order of cases does not matter

```verilog
// Simple encoder
module encode (A, Y);
input  [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
  case (A)
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3'bX; // Don’t care when input is all 0’s
  endcase
endmodule
```
Verilog casex

- Like case, but cases can include ‘X’
  - X bits not used when evaluating the cases
  - In other words, you don’t care about those bits!

Verilog for

- `for` is similar to C
- `for` statement is executed at compile time (like macro expansion)
  - Result is all that matters, not how result is calculated
  - Use in testbenches only!

Another Behavioral Example

- Computing Conway’s Game of Life rule
  - Cell with no neighbors or 4 neighbors dies; with 2-3 neighbors lives

```verilog
module life (neighbors, self, out);
  input         self;
  input [7:0]   neighbors;
  output        out;
  reg out;
  integer       count;
  integer       i;
  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i + 1) begin
      if (neighbors[i] == 1) count = count + neighbors[i];
      if (self == 1) count = count + 1;
    end
    out = count == 3;
  end
endmodule
```
**Verilog while/repeat/forever**

- while (expression) statement
  - Execute statement while expression is true
- repeat (expression) statement
  - Execute statement a fixed number of times
- forever statement
  - Execute statement forever

**full-case and parallel-case**

- // synthesis parallel_case
  - Tells compiler that ordering of cases is not important
  - That is, cases do not overlap
    - e.g., state machine - can’t be in multiple states
  - Gives cheaper implementation
- // synthesis full_case
  - Tells compiler that cases left out can be treated as don’t cares
  - Avoids incomplete specification and resulting latches

**Sequential Logic**

//Parallel to Serial converter
module ParToSer(LD, X, out, CLK);
  input [3:0] X;
  input LD, CLK;
  reg out;
  assign out = Q[0];
  always @ (posedge CLK)
    assign B = {1'b0, A[3:1]}
endmodule // ParToSer

Testbench

Top-level modules written specifically to test sub-modules.

General no ports.

- Notes:
  - initial block similar to always except only executes once (at beginning of simulation)
  - #n’s needed to advance time
  - $monitor - prints output

```verilog
module testmux;
  reg a, b, s;
  wire f;
  reg expected;
 mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));
  initial
    a=0; b=0; s=1; expected=0;
    f=1; $monitor( "select=0\n in0=0\n in1=1\n out=1\n expected=0\n time=1",
 s, a, b, f, expected, $time);
endmodule // testmux
```

Notes: A variety of other “system functions”, similar to $monitor exist for displaying output and controlling the simulation.
Final thoughts on Verilog Examples

• Verilog looks like C, but it describes hardware
  – Multiple physical elements, Parallel activities
  – Temporal relationships
  – Basis for simulation and synthesis
  – figure out the circuit you want, then figure out how to express it in Verilog

• Understand the elements of the language
  – Modules, ports, wires, reg, primitive, continuous assignment, blocking statements, sensitivity lists, hierarchy
  – Best done through experience

• Behavioral constructs hide a lot of the circuit details but you as the designer must still manage the structure, data-communication, parallelism, and timing of your design.