CIS 410/510
HDL: Verilog Fundamentals
Part I

Prof. Michel A. Kinsy
Opportunities for exceptional projects

- Possibility of targeting FPGA
  - Various development boards are available
  - Requires different toolchains
    - Xilinx Vivado
    - Altera Quartus II
Verilog Fundamentals

• **Data types**

• Structural Verilog

• Functional Verilog
  ‣ Gate level
  ‣ Register transfer level
  ‣ High-level behavioral
Primary Verilog data type

- Primary Verilog data type is a bit-vector where bits can take

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero</td>
</tr>
<tr>
<td>1</td>
<td>Logic one</td>
</tr>
<tr>
<td>X</td>
<td>Unknown logic value</td>
</tr>
<tr>
<td>Z</td>
<td>High impedance, floating</td>
</tr>
</tbody>
</table>

An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.
• The Verilog keyword **wire** is used to denote a standard hardware net

```verilog
wire [15:0] instruction;
wire [15:0] memory_req;
wire [ 7:0] small_net;
```

Absolutely no type safety when connecting nets!
Verilog bit literals

- Verilog includes ways to specify bit literals in various bases

- Binary literals
  - 8’b0000_0000
  - 8’b0xx0_1xx1

- Hexadecimal literals
  - 32’h0a34_def1
  - 16’haxxx

- Decimal literals
  - 32’d42
Verilog Fundamentals

• Data types

• **Structural Verilog**

• Functional Verilog
  ‣ Gate level
  ‣ Register transfer level
  ‣ High-level behavioral
A Verilog module includes a module name and a port list.

```verilog
class module adder( A, B, cout, sum );
  input [3:0] A;
  input [3:0] B;
  output cout;
  output [3:0] sum;
// HDL modeling of
// adder functionality
endmodule
```

Ports must have a direction (or be bidirectional) and a bitwidth.

Note the semicolon at the end of the port list!
Verilog module specification

- A Verilog module includes a module name and a port list.

**Traditional Verilog-1995 Syntax**

```verilog
module adder( A, B, cout, sum );
    input  [3:0] A;
    input  [3:0] B;
    output cout;
    output [3:0] sum;
endmodule
```

**ANSI C Style Verilog-2001 Syntax**

```verilog
module adder( input [3:0] A,
             input [3:0] B,
             output cout,
             output [3:0] sum );
endmodule
```
Module composition

• A module can instantiate other modules creating a module

```
module FA( input a, b, cin 
    output cout, 
    sum );

    // HDL modeling of 1 bit 
    // adder functionality

dendmodule
```
module adder( input [3:0] A, B, output [3:0] cout, output [3:0] S );

wire c0, c1, c2;
FA fa0( ... );
FA fa1( ... );
FA fa2( ... );
FA fa3( ... );
endmodule
module adder( input [3:0] A, B, 
output cout, 
output [3:0] S );

wire c0, c1, c2;
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
FA fa1( A[1], B[1], c0, c1, S[1] );
FA fa2( A[2], B[2], c1, c2, S[2] );
FA fa3( A[3], B[3], c2, cout, S[3] );
endmodule
Module Composition

- Verilog supports connecting ports by position and by name

  Connecting ports by ordered list

  ```verilog
  FA fa0( A[0], B[0], 1'b0, c0, S[0] );
  ```

  Connecting ports by name (compact)

  ```verilog
  FA fa0( .a(A[0]), .b(B[0]),
          .cin(1'b0), .cout(c0), .sum(S[0]) );
  ```

  Connecting ports by name

  ```verilog
  FA fa0
  (    .a    (A[0]),
    .b    (B[0]),
    .cin  (1'b0),
    .cout (c0),
    .sum  (S[0])
  );
  ```

  For all but the smallest modules, connecting ports by name yields clearer and less buggy code.
Verilog Fundamentals

• Data types
• Structural Verilog
• **Functional Verilog**
  ‣ Gate level
  ‣ Register transfer level
  ‣ High-level behavioral
Functional Verilog

- Functional Verilog can roughly be divided into three abstraction levels

  - **Behavioral Algorithm**
    - Manual
    - Abstract algorithmic description
      - Describes how data flows between state elements for each cycle
  
  - **Register Transfer Level**
    - Logic Synthesis
    - Low-level netlist of primitive gates
  
  - **Gate Level**
    - Auto Place + Route

module mux4( input a, b, c, d, input [1:0] sel, output out );

wire [1:0] sel_b;
not not0( sel_b[0], sel[0] );
not not1( sel_b[1], sel[1] );

wire n0, n1, n2, n3;
and and0( n0, c, sel[1] );
and and1( n1, a, sel_b[1] );
and and2( n2, d, sel[1] );
and and3( n3, b, sel_b[1] );

wire x0, x1;
nor nor0( x0, n0, n1 );
nor nor1( x1, n2, n3 );

wire y0, y1;
or or0( y0, x0, sel[0] );
or or1( y1, x1, sel_b[0] );
nand nand0( out, y0, y1 );

endmodule
Continuous Assignments

• Continuous assignment statements assign one net to another or to a literal

Explicit continuous assignment

```vhdl
wire [15:0] netA;
wire [15:0] netB;
assign netA = 16'h3333;
assign netB = netA;
```

Implicit continuous assignment

```vhdl
wire [15:0] netA = 16'h3333;
wire [15:0] netB = netA;
```
Continuous Assignments

• Using continuous assignments to implement an RTL four input multiplexer

```verilog
module mux4( input a, b, c, d
             input [1:0] sel,
             output out );

wire out, t0, t1;

assign t0 = ~( (sel[1] & c) | (~sel[1] & a) );
assign t1 = ~( (sel[1] & d) | (~sel[1] & b) );
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule
```

The order of these continuous assignment statements does not matter. They essentially happen in parallel!
Other Verilog Operators

• Verilog RTL includes many operators in addition to basic boolean logic

```verilog
// Four input multiplexer
module mux4( input a, b, c, d
             input [1:0] sel,
             output out );

    assign out = ( sel == 0 ) ? a :
            ( sel == 1 ) ? b :
            ( sel == 2 ) ? c :
            ( sel == 3 ) ? d : 1'b0;

endmodule

// Simple four bit adder
module adder( input [3:0] op1, op2,
              output [3:0] sum );

    assign sum = op1 + op2;

endmodule
```

If input is undefined we want to propagate that information
Verilog RTL operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>+ - * / % **</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>! &amp;&amp;</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt; &lt; &gt;= &lt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>== != === !==</td>
</tr>
<tr>
<td>Bitwise</td>
<td>~ &amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduction</th>
<th>&amp; ~&amp;</th>
<th>~</th>
<th>^ ^~</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;&gt;</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{ }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

wire [ 3:0] net1 = 4'b00xx;
wire [ 3:0] net2 = 4'b1110;
wire [11:0] net3 = { 4'b0, net1, net2 };
wire equal = ( net3 === 12'b0000_1110_00xx );

Avoid (/ % **) since the usually synthesize poorly
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4( input a, b, c, d
           input [1:0] sel,
           output out );

reg out, t0, t1;

always @( a or b or c or d or sel )
begin
  t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  out = ~( (t0 | sel[0]) & (t1 | ~sel
end

endmodule
```

The always block is reevaluated whenever a signal in its sensitivity list changes.
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

reg out, t0, t1;

always @( a or b or c or d or sel )
begin
  t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  out = ~( (t0 | sel[0]));
end

endmodule
```

The order of these procedural assignment statements does matter. They essentially happen in sequentially!
Procedural Assignments

• **Always blocks** have parallel inter-block and sequential intra-block semantics

```verilog
modulemux4(inputa, b, c, d
           input[1:0]sel,
           outputout);

regout, t0, t1;

always @(a or b or c or d or sel)
begin
    t0 = ~(sel[1] & c) | (~sel[1] & a);
    t1 = ~(sel[1] & d) | (~sel[1] & b);
    out = ~(t0 | sel[0]) & (t1 | ~sel[0]);
end

endmodule
```

LHS of procedural assignments must be declared as a reg type. Verilog reg is not necessarily a hardware register!
Procedural Assignments

- Always blocks have parallel inter-block and sequential intra-block semantics

```verilog
defmodule mux4 (input a, b, c, d
   input [1:0] sel,
   output out );

   reg out, t0, t1;

   always @( a or b or c or d or sel )
   begin
      t0 = ~( (sel[1] & c) | (~sel[1] & a) );
      t1 = ~( (sel[1] & d) | (~sel[1] & b) );
      out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
   end

endmodule
```

What happens if we accidentally forget a signal on the sensitivity list?
Procedural Assignments

- **Always blocks** have parallel inter-block and sequential intra-block semantics

```verilog
testbench
testbench
```

Verilog-2001 provides special syntax to automatically create a sensitivity list for all signals read in the always block.

```verilog
module mux4( input a, b, c, d
            input [1:0] sel,
            output out );

    reg out, t0, t1;

    always @( * )
    begin
        t0  = ~( (sel[1] & c) | (~sel[1] & a) );
        t1  = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule
```
Next Class

• Part II