Verilog 1 - Fundamentals

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
What is Verilog?

- Verilog is a hardware description language
  - There is more to it than that, but this is what we will use it for.

- In this class, Verilog is an implementation language, not a design language
  - When you sit down to write verilog you should know exactly what you are implementing.
  - Draw your schematic and state machines and transcribe it into Verilog.

- We are constraining you to a subset of the language for two reasons
  - Reduce complexity for the course
  - These are parts that people use to design real processors

Courtesy of Arvind
http://csg.csail.mit.edu/6.375/
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors

```verilog
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
```

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Bit-vector is the only data type in Verilog

A bit can take on one of four values

<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.

In the simulation waveform viewer, Unknown signals are **RED**. There should be no red after reset.

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
“wire” is used to denote a hardware net

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

Absolutely no type safety when connecting nets!
Bit literals

4'b10_11

Binary literals
- 8'b0000_0000
- 8'b0xx0_1xx1

Hexadecimal literals
- 32'h0a34_def1
- 16'haxxx

Decimal literals
- 32'd42

Underscores are ignored
Base format (d,b,o,h)
Decimal number representing size in bits

We’ll learn how to actually assign literals to nets a little later

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors

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
Our Verilog Subset

- Verilog is a big language with many features not concerned with synthesizing hardware.
- The code you write for your processor should only contain the languages structures discussed in these slides.
- Anything else is not synthesizable, although it will simulate fine.
- You *MUST* follow the course coding standard

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
A Verilog module has a name and a port list

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

// HDL modeling of // adder functionality
endmodule

Note the semicolon at the end of the port list!

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

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
A module can instantiate other modules

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

module FA(input a, b, cin, output cout, sum);
// HDL modeling of 1 bit full adder functionality
endmodule
```

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
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
Alternative syntax

Connecting ports by ordered list

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

Connecting ports by name (compact)

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

Argument order does not matter when ports are connected by name

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

Connecting ports by name yields clearer and less buggy code.
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- Simple behaviors

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

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
A module’s behavior can be described in many different ways but it should not matter from outside

Example: mux4
mux4: Using continuous assignments

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

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

endmodule
```

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

Language defined operators
mux4: Behavioral style

// 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'bX;
endmodule

If input is undefined we want to propagate that information.
mux4: Using “always block”

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

The order of these procedural assignment statements DOES matter. They essentially happen sequentially!
"Always blocks" permit more advanced sequential idioms

module mux4( input a, b, c, d, input [1:0] sel, output out );
reg out;
always @( * )
bEGIN
if ( sel == 2’d0 )
out = a;
else if ( sel == 2’d1 )
out = b
else if ( sel == 2’d2 )
out = c
else if ( sel == 2’d3 )
out = d
else
out = 1’bx;
end
endmodule

module mux4( input a, b, c, d, input [1:0] sel, output out );
reg out;
always @( * )
bEGIN
  case ( sel )
    2’d0 : out = a;
    2’d1 : out = b;
    2’d2 : out = c;
    2’d3 : out = d;
    default : out = 1’bx;
  endcase
end
endmodule

Typically we will use always blocks only to describe sequential circuits.
What happens if the case statement is not complete?

```verilog
module mux3( input a, b, c,
            input [1:0] sel,
            output out );

reg out;
always @( * )
begin
    case ( sel )
        2’d0 : out = a;
        2’d1 : out = b;
        2’d2 : out = c;
    endcase
end
endmodule
```

If sel = 3, mux will output the previous value!

What have we created?
What happens if the case statement is not complete?

module mux3( input a, b, c,
            input [1:0] sel,
            output out );

reg out;

always @( * )
begin
    case ( sel )
        2’d0 : out = a;
        2’d1 : out = b;
        2’d2 : out = c;
        default : out = 1’bx;
    endcase
end
endmodule

We CAN prevent creating state with a default statement

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Parameterized mux4

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

wire [WIDTH-1:0] out, t0, t1;

assign t0 = (sel[1]? c : a);
assign t1 = (sel[1]? d : b);
assign out = (sel[0]? t0: t1);
endmodule
```

Instantiation Syntax

```
mux4#(32) alu_mux
(.a (op1),
 .b (op2),
 .c (op3),
 .d (op4),
 .sel (alu_mux_sel),
 .out (alu_mux_out) );
```

Parameterization is a good practice for reusable modules
Writing a muxn is challenging
Verilog Registers “reg”

- Wires are line names – they do not represent storage and can be assigned only once.

- Regs are imperative variables (as in C):
  - “nonblocking” assignment: $r <= v$
  - can be assigned multiple times and holds values between assignments.
flip-flops

module FF0 (input clk, input d, output reg q);
always @( posedge clk )
begin
    q <= d;
end
endmodule

module FF (input clk, input d, input en, output reg q);
always @( posedge clk )
begin
    if ( en )
        q <= d;
end
endmodule
flip-flops with reset

```verilog
always @(posedge clk)
begin
  if (!resetN)
    Q <= 0;
  else if (enable)
    Q <= D;
end
```

synchronous reset
module register#(parameter WIDTH = 1)
(
    input clk,
    input [WIDTH-1:0] d,
    input en,
    output [WIDTH-1:0] q
);

always @(posedge clk) 
begin
    if (en)
        q <= d;
end
endmodule
Register in terms of Flipflops

module register2
(input clk,
 input [1:0] d,
 input en,
 output [1:0] q);

always @(posedge clk)
begin
  if (en)
    q <= d;
end
endmodule

Do they behave the same?

yes