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
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

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.
“wire” is used to denote a hardware net

```verilog
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

```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
```
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

module adder(input [3:0] A, B, cout, output [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
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
```
A module’s behavior can be described in many different ways but it should not matter from outside.

Example: mux4
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!

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
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.
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):
  - can be assigned multiple times and holds values between assignments.
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!

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

Typically we will use always blocks only to describe sequential circuits
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;
    endcase
end

endmodule

What have we created?

If sel = 3, mux will output the previous value!
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.
Parameterized mux4

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

Parameterization is a good practice for reusable modules
Writing a muxn is challenging

Instantiation Syntax

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

- Inside an always block, '=' is a blocking assignment.
  - The assignment happens immediately and affects subsequent statements in the always block.
- '<=' is a non-blocking assignment.
  - All the assignments happen at the end of the block.
Non-blocking example

Initially, $a = 2$, $b = 3$

```verilog
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(posedge clock)
begin
  a <= b;
  c <= a;
and
Afterwards: $a = 3$ and $c = 2$
```

```verilog
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(*)
begin
  a = b;
  c = a;
and
Afterwards: $a = 3$ and $c = 3$
```
flip-flops

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

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

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

module register2
(input clk,
 input [1:0] d,
 input en,
 output [1:0] q);
  FF ff0 (.clk(clk), .d(d[0]), .en(en), .q(q[0]));
  FF ff1 (.clk(clk), .d(d[1]), .en(en), .q(q[1]));
endmodule

Do they behave the same?
yes
Verilog can be used at several levels

A common approach is to use C/C++ for initial behavioral modeling, and for building test rigs.

Automatic tools to synthesize a low-level gate-level model.
Writing synthesizable Verilog: Combinational logic

- Use continuous assignments (`assign`)
  ```verilog
  assign C_in = B_out + 1;
  ```
- Use `always@(*)` blocks with blocking assignments (`=`)
  ```verilog
  always @(*)
  begin
    out = 2'd0;
    if (in1 == 1)
      out = 2'd1;
    else if (in2 == 1)
      out = 2'd2;
  end
  ```

Always blocks allow more expressive control structures, though not all will synthesize default value.

- Every variable should have a default value to avoid inadvertent introduction of latches
- Do not assign the same variable from more than one always block – ill defined semantics
Writing synthesizable Verilog: Sequential logic

- Use `always @(posedge clk)` and non-blocking assignments (`<=`)

```verilog
class @ (posedge clk)
C_out <= C_in;
```

- Use only positive-edge triggered flip-flops for state

- Do not assign the same variable from more than one `always` block – ill defined semantics

- Do not mix blocking and non-blocking assignments

- Only leaf modules should have functionality; use higher-level modules only for wiring together sub-modules
Declaring register files

module reg_file
(
    input clk, // Global clock
    input reset, // Global reset
    input [1: 0] r1, // Encoded register inputs
    input [31: 0] r_data,
    input write_en,
    output [31: 0] v1
);

reg [31:0] reg_file [3:0];

assign v1 = reg_file[r1];

always @(posedge clk)
begin
    if(reset)
        begin // Do not use this anywhere else.
            reg_file[0] <= 0;
            reg_file[1] <= 0;
            reg_file[2] <= 0;
            reg_file[3] <= 0;
        end
    else
        begin
            if(write_en)
                reg_file[r1] <= r_data;
        end
end
endmodule