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?

In this class and in the real world, Verilog is a specification language, not a programming language.

- Draw your schematic and state machines and then transcribe it into Verilog.
- When you sit down to write verilog you should know exactly what you are implementing.

We are constraining you to a subset of the language for two reasons

- These are the parts that people use to design real processors
- Steer you clear of problematic constructs that lead to bad design.
Verilog Fundamentals

- What is Verilog?
- Data types
- Structural Verilog
- RTL Verilog
  - Combinational Logic
  - Sequential Logic

```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
```
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

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

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

Binary literals
- 8'b0000_0000
- 8'b0xx0_1xx1

Hexadecimal literals
- 32'h0a34_def1
- 16'haxxx

Decimal literals
- 32'd42
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- RTL 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
```
Note: 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; a document will be released soon.
- We will be mixing in some synthesizable SystemVerilog later in the course to improve maintainability of your code.
A Verilog module has a name and a port list

```verilog
module adder( input [3:0] a_i, input [3:0] b_i, output cy_o, output [3:0] sum_o );

// HDL modeling of // adder functionality
endmodule
```

Ports must have a direction and a bitwidth. In this class we use _i to denote in port variables and _o to denote out port variables.

Note the semicolon at the end of the port list!
A module can instantiate other modules

module adder( input [3:0] a_i, b_i, output cy_o, output [3:0] sum_o );
wire c0, c1, c2;
FA fa0( ... );
FA fa1( ... );
FA fa2( ... );
FA fa3( ... );
endmodule

module FA( input a_i, b_i, cy_i output cy_o, sum_o );
// HDL modeling of 1 bit
// full adder functionality
endmodule
module adder( input [3:0] a_i, b_i, 
    output cy_o, 
    output [3:0] sum_o );

wire c0, c1, c2;
FA fa0( a_i[0], b_i[0], 1'b0, c0, sum_o[0] );
FA fa1( a_i[1], b_i[1], c0, c1, sum_o[1] );
FA fa2( a_i[2], b_i[2], c1, c2, sum_o[2] );
FA fa3( a_i[3], b_i[3], c2, cy_o, sum_o[3] );
endmodule
This class’s style standard:
Connect ports by name and not by position.

Connecting ports by ordered list is compact but bug prone:

```vhdl
FA fa0( a_i[0], b_i[0], 1'b0, c0, sum_o[0] );
```

Connecting by name is less compact but leads to fewer bugs. This is how you should do it in this class. You should also line up like parameters so it is easy to check correctness.

```vhdl
FA fa0( .a_i(a_i[0])
    ,.b_i(b_i[0])
    ,.cy_i(1'b0)
    ,.cy_o(c0)
    ,.sum_o(sum_o[0])
);
```

Connecting ports by name yields clearer and less buggy code. In the slides, we may do it by position for space. But you should do it by name and not position.
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- RTL
  - Combinational
  - Sequential

```verilog
module adder(
    input [3:0] A, B,
    output c0, c1, c2,
    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, c0, 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_i, b_i, c_i, d_i,
    input [1:0] sel_i,
    output z_o);

wire t0, t1;

assign z_o = ~((t0 | sel_i[0]) & (t1 | ~sel_i[0]));
assign t1  = ~((sel_i[1] & d_i) | (~sel_i[1] & b_i));
assign t0  = ~((sel_i[1] & c_i) | (~sel_i[1] & a_i));

endmodule

mux4: Using continuous assignments to generate combinational logic

The order of these continuous assignment statements in the source code does not matter. But it does affect readability!

They essentially happen in parallel; also, any time an input is changed, each line is automatically re-evaluated. (Be careful not to create cycles!)
mux4: Using ? :

// Four input multiplexer
module mux4( input a_i, b_i, c_i, d_i,
             input [1:0] sel_i,
             output z_o);

assign z_o = ( sel_i == 0 ) ? a_i :
             ( sel_i == 1 ) ? b_i :
             ( sel_i == 2 ) ? c_i :
             ( sel_i == 3 ) ? d_i : 1'bx;
endmodule

Not required for synthesis, but helps in simulation: If sel_i is undefined we want to propagate that information in waveform viewer.
mux4: Using combinational “always_comb” or “always @(*)” block

```
module mux4( input  a_i, b_i, c_i, d_i,
               input [1:0] sel_i,
               output reg  z_o );

   reg t0, t1;

   always_comb // system verilog; equiv. to always @(*)
   begin
      t0  =  (sel_i[1] & c_i) | (~sel_i[1] & a_i);
      t1  =  ~(sel_i[1] & d_i) | (~sel_i[1] & b_i));
      t0  =  ~t0;
      z_o  =  ~( (t0 | sel_i[0]) & (t1 | ~sel_i[0]) )
   end
endmodule
```

Within the always @(*) begin/end block, effects of statements appear to execute sequentially; Outside of block, only the last assignment to each variable is visible, and it appears a short time after any input is changed.
For instance, the second t0 line uses t0 from the first.
“Always @(*)” permit more advanced combinational idioms

module mux4( input a_i,b_i,c_i,d_i
           input [1:0] sel_i,
           output reg z_o);

always_comb
begin
  if (sel_i == 2’d0 )
    z_o = a_i;
  else if (sel_i == 2’d1)
    z_o = b_i;
  else if (sel_i == 2’d2)
    z_o = c_i;
  else if (sel_i == 2’d3)
    z_o = d_i;
  else
    z_o = 1’bx;
end
endmodule
What happens if the case statement is not complete?

```verilog
module mux3( input a_i, b_i, c_i,
             input [1:0] sel_i,
             output reg z_o );

always @( * )
begin
    case ( sel_i )
        2’d0 : z_o = a_i;  
        2’d1 : z_o = b_i;
        2’d2 : z_o = c_i;
    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?

```verilog
module mux3( input a_i, b_i, c_i
       input [1:0] sel_i,
       output reg z_o );

always @( * )

begin

    case ( sel_i )
        2'd0 : z_o = a_i;
        2'd1 : z_o = b_i;
        2'd2 : z_o = c_i;
        default : z_o = 1'bx;
    endcase

end

endmodule

We CAN prevent creating a latch with a default statement
```
Parameterized mux4

```verilog
module mux4 #(parameter WIDTH = 1 )
  (input [WIDTH-1:0] a_i, b_i, c_i, d_i,
   input [1:0] sel_i,
   output [WIDTH-1:0] z_o);

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

assign t0  = (sel_i[1]? c_i : a_i);
assign t1  = (sel_i[1]? d_i : b_i);
assign z_o = (sel_i[0]? t0: t1);
endmodule
```

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

Instantiation
```
mux4#(32) alu_mux
  (.a_i (op1),
   .b_i (op2),
   .c_i (op3),
   .d_i (op4),
   .sel_i(alu_mux_sel),
   .z_o(alu_mux_out));
```
Verilog Fundamentals

- History of hardware design languages
- Data types
- Structural Verilog
- RTL
  - Combinational
  - Sequential

```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
```
Sequential Logic: Creating a flip flop

```vhdl
reg q_r, q_next;
always_ff @(posedge clk)
  q_r <= q_next;
```

1) The keyword `reg` confusingly enough does not have much to do with registers; it's just used to indicate a wire that is driven from a `always_ff` or `always_comb` block. So this line simply creates two wires, one called `q_r` and the other called `q_next`.

2) `always_ff` keyword indicates our intent to create registers; you can use the `always` keyword instead, but then the synthesizer has to guess!

3) `@ (posedge clk )` indicates that we want these registers to be triggered on the positive edge of the `clk` clock signal.

4) Combined with 2) and 3), the `<=` creates a register whose input is wired to `q_next` and whose output is wired to `q_r`.

Note: always use `<=` with `always_ff` and `=` with `always_comb`
Sequential Logic: flip-flop variants

module FF0 (input clk, input d_i, output reg q_r_o);

always_ff @(posedge clk)
begin
    q_r_o <= d_i;
end
endmodule

module FF (input clk, input d_i, input en_i, output reg q_r_o);

always_ff @(posedge clk)
begin
    if (en_i)
        q_r_o <= d_i;
end
endmodule
flip-flops with reset

```verilog
always_ff @(posedge clk)
begin
    if (reset)
        Q <= 0;
    else if (enable)
        Q <= D;
end
```

synchronous reset
Register (i.e. a vector of flip-flops)

module register#(parameter WIDTH = 1) 
(
    input clk, 
    input [WIDTH-1:0] d_i, 
    input en_i, 
    output reg [WIDTH-1:0] q_r_o 
);

always_ff @(posedge clk )
begin
    if (en_i)
        q_r_o <= d_i;
end
endmodule
Implementing Wider Registers

module register2
(input clk,
 input [1:0] d_i,
 input en_i,
 output reg [1:0] q_r_o
);

always_ff @(posedge clk)
begin
    if (en_i)
        q_r_o <= d_i;
end
endmodule

Do they behave the same?

yes
Syntactic Sugar: `always_ff` allows you to combine combinational and sequential logic; *but this can be confusing.*

**more clear**

```verilog
module accum
  ( input clk,
    input data_i,
    input en_i,
    output [3:0] sum_o;
  );
  reg [WIDTH-1:0] sum_r, sum_next;
  assign sum_o = sum_r;

  always_comb
  begin
    sum_next = sum_r;
    if (en_i) sum_next = sum_r + data_i;
  end

  always_ff @(posedge clk)
  sum_r <= sum_next;

endmodule
```

**shorter**

```verilog
module accum
  ( input clk,
    input data_i,
    input en_i,
    output [3:0] sum_o;
  );
  reg [WIDTH-1:0] sum_r;
  assign sum_o = sum_r;

  always_ff @(posedge clk)
  begin
    if (en_i) sum_r <= sum_r + data_i;
  end

endmodule
```
Syntactic Sugar: You can always convert an `always_ff` that combines combinational and sequential logic into two separate `always_ff` and `always_comb` blocks.

**shorter**

```verilog
module accum
    (input clk,
     input data_i,
     input en_i,
     output [3:0] sum_o);

    reg [WIDTH-1:0] sum_r;
    assign sum_o = sum_r;

    always_ff @(posedge clk)
    begin
        if (en_i)
            sum_r <= sum_r + data_i;
    end

endmodule
```

**more clear**

```verilog
module accum
    (input clk,
     input data_i,
     input en_i,
     output [3:0] sum_o);

    reg [WIDTH-1:0] sum_r, sum_next;
    assign sum_o = sum_r;

    always_comb
    begin
        sum_next = sum_r;
        if (en_i)
            sum_next = sum_r + data_i;
    end

    always_ff @(posedge clk)
    sum_r <= sum_next;

endmodule
```

When in doubt, use the version on the right.

To go from the left-hand version to the right one:

1. For each register `xxx_r`, introduce a temporary variable that holds the input to each register (e.g. `xxx_next`)
2. Extract the combinational part of the `always_ff` block into an `always_comb` block:
   a. change `xxx_r <=` to `xxx_next =`
   b. add `xxx_next = xxx_r;` to beginning of block for default case
3. Extract the sequential part of the `always_ff` by creating a separate `always_ff` that does `xxx_r <= xxx_next;`
Bit Manipulations

```verilog
wire [15:0] x;
wire [31:0] x_sext;
wire [31:0] hi, lo;
wire [63:0] hilo;

// concatenation
assign hilo = { hi, lo };
assign { hi, lo } = { 32'b0, 32'b1 };

// duplicate bits (16 copies of x[15] + x[15:0])
assign x_sext = {16 { x[15] }, x[15:0]};
```