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
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
```
Originally designers used breadboards for prototyping.

Solderless Breadboard

No symbolic execution or testing

Printed circuit board

Number of Gates in Design

10^7
10^6
10^5
10^4
10^3
10^2
10

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
HDLs enabled logic level simulation and testing

Gate Level Description

Manual

Simulate

Test Results

Number of Gates in Design

10^7
10^6
10^5
10^4
10^3
10^2
10

HDL = Hardware Description Language

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Designers began to use HDLs for higher level design.

- Behavioral Algorithm
- Register Transfer Level
- Gate Level
- Manual

Simulate → Test Results

Number of Gates in Design:
- $10^2$
- $10^3$
- $10^4$
- $10^5$
- $10^6$
- $10^7$

 Courtesy of Arvind http://csg.csail.mit.edu/6.375/
HDLs led to tools for automatic translation

- Behavioral Algorithm
  - Manual
- Register Transfer Level
- Logic Synthesis
- Gate Level
- Auto Place + Route

Number of Gates in Design

<table>
<thead>
<tr>
<th>Number of Gates in Design</th>
<th>10^7</th>
<th>10^6</th>
<th>10^5</th>
<th>10^4</th>
<th>10^3</th>
<th>10^2</th>
<th>10</th>
</tr>
</thead>
</table>

HDLs: Verilog, VHDL...
Tools: Spice, ModelSim, DesignCompiler, ...

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Raising the abstraction further ...

Guarded Atomic Actions
GAA Compiler
Register Transfer Level
Logic Synthesis
Gate Level
Auto Place + Route

Number of Gates in Design

10^7
10^6
10^5
10^4
10^3
10^2
10

Simulate
Test Results

Simulate
Test Results

Simulate
Test Results

Simulate
Test Results

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
The current situation

C, C++, SystemC

Behavioral RTL
Verilog, VHDL, SystemVerilog

MATLAB

Structural RTL
Verilog, VHDL, SystemVerilog

Register Transfer Level

Logic Synthesis

Simulators and other tools are available at all levels but not compilers from the behavioral level to RTL

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
```
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.

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

- 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

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/
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.

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

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

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/
Alternate syntax

Traditional Verilog-1995 Syntax

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

ANSI C Style Verilog-2001 Syntax

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

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

Connecting ports by name (compact)

```vhdl
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

```vhdl
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

```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
```
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 [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

mux4: Gate-level structural Verilog

b d a c sel[1] sel[0]
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) ) ;

demodule

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.
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?

```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?

```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;
        default : out = 1'b0;
    endcase
end

endmodule
```

We CAN prevent creating state with a default statement.
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

```verilog
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 mux\( n \) 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

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
flip-flops with reset

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

What is the difference?

synchronous reset

always @(posedge clk or negedge resetN)
begin
    if (~resetN)
        Q <= 0;
    else if (enable)
        Q <= D;
end

asynchronous reset
Latches versus flip-flops

module latch
(
    input clk,
    input d,
    output reg q
);

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

module flipflop
(
    input clk,
    input d,
    output reg q
);

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

Edge-triggered always block

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

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
Three abstraction levels for functional descriptions

1. Behavioral Algorithm
   - Manual
   - Describes how data flows between state elements for each cycle

2. Register Transfer Level
   - Logic Synthesis
   - Low-level netlist of primitive gates

3. Gate Level
   - Auto Place + Route

Next time
Some examples

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