Dataflow Verilog
Motivation

• Structural design can be cumbersome
  − Lots of typing
    • 32-bit busses & logic

• Structural designs are static
  − At least in Verilog (not so for VHDL)
  − Little or no parameterization possible
module mux21(q, sel, a, b);
    input sel, a, b;
    output q;

    assign q = (~sel & a) | (sel & b);
endmodule

Much simpler, less typing, familiar C-like syntax. Synthesizer turns it into optimized gate-level design.
A Dataflow MUX - Version 2

module mux21(q, sel, a, b);
    input sel, a, b;
    output q;

    assign q = sel ? b : a;
endmodule

Even simpler, uses C-like ternary (? : ) construct
## Dataflow Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
<th># of Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*, /, +, -</td>
<td>As expected</td>
<td>2</td>
<td>* and / take LOTS of hardware</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Modulo</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>Logic NOT</td>
<td>1</td>
<td>As in C</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>Logic AND</td>
<td>2</td>
<td>As in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logic OR</td>
</tr>
<tr>
<td>Bitwise</td>
<td>~</td>
<td>Bitwise NOT</td>
<td>1</td>
<td>As in C</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>2</td>
<td>As in C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bitwise OR</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>Bitwise XOR</td>
<td>2</td>
<td>As in C</td>
</tr>
<tr>
<td></td>
<td>~^</td>
<td>Bitwise XNOR</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>As expected</td>
<td>2</td>
<td>As in C</td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=</td>
<td>As expected</td>
<td>2</td>
<td>As in C</td>
</tr>
<tr>
<td>Reduction</td>
<td>&amp;</td>
<td>Red. AND</td>
<td>1</td>
<td>Multi-bit input</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>Red. NAND</td>
<td>1</td>
<td>Multi-bit input</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Red. OR</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
<td>Red. NOR</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>Red. XOR</td>
<td>1</td>
<td>Multi-bit input</td>
</tr>
<tr>
<td></td>
<td>~^</td>
<td>Red. XNOR</td>
<td>1</td>
<td>Multi-bit input</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;</td>
<td>Left shift</td>
<td>2</td>
<td>Fill with 0's</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>Right shift</td>
<td>2</td>
<td>Fill with 0's</td>
</tr>
<tr>
<td>Concat</td>
<td>{ }</td>
<td>Concatenate</td>
<td>Any number</td>
<td></td>
</tr>
<tr>
<td>Replicate</td>
<td>{ { } }</td>
<td>Replicate</td>
<td>Any number</td>
<td></td>
</tr>
<tr>
<td>Cond</td>
<td>?:</td>
<td>As expected</td>
<td>3</td>
<td>As in C</td>
</tr>
</tbody>
</table>
Bitwise vs. Logic Operators

• Similar to C

```verilog
assign q = ((a<4'b1101) && ((c&4'b0011)!=0)) ? 1'b0 : 1'b1;
```

Pseudo-code (not real Verilog):
```
if (a<4'b1101 && (c&4'b0011) != 0) then
  q <= '0';
else
  q <= '1';
```

Use `&&`, `||`, `!` for 1-bit quantities (results of comparisons)

Use `&`, `|`, `~` for bit-by-bit logical operations
Reduction Operators

wire[3:0] x;
wire z;

Concatenation and Replication Operators

wire[3:0] x, y;
wire[7:0] z, q, w, t;
wire[31:0] m, n;

assign x = 4'b1100;
assign y = 4'b0101;
assign z = {x, x};   // z is 8'b11001100
assign q = {x, y};   // q is 8'b11000101
assign w = {4'b1101, y};   // w is 8'b11010101
assign t = {2{x}};   // same as {x, x}
assign m = {{4{x}}, {2{q}}};
  // m is 32'b11001100110011001100010111000101
Operator Precedence

- Similar to C

<table>
<thead>
<tr>
<th>Higher precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary -, unary +, !, ~</td>
</tr>
<tr>
<td>*, /, %</td>
</tr>
<tr>
<td>+, -</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>==, !=</td>
</tr>
<tr>
<td>&amp;, ~&amp;</td>
</tr>
<tr>
<td>^, ~^</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>?::</td>
</tr>
</tbody>
</table>

Lower precedence
A Note on Matching Widths

• This a valid 2:1 MUX statement:
  
  wire a, b, sel, q;
  assign q = (~sel & a) | (sel & b);

• But the following is not:
  
  wire[3:0] a, b, q;
  wire sel;
  assign q = (~sel & a) | (sel & b);

Why?
More On Matching Wire Widths

• This is an acceptable substitute:

```verilog
wire[3:0] a, b, q;
wire sel;
assign q = ({4{~sel}}&a)|({4{sel}}&b);
```

• It turns the sel and ~sel values into 4-bit versions for AND-ing and OR-ing

• A more elegant version:

```verilog
wire[3:0] a, b, q;
wire sel;
assign q = sel ? b: a;
```
Design Example: A 2:4 Decoder

module decode24(q, a);
    output[3:0] q;
    input[1:0] a;

    assign q = (4'b0001) << a;
endmodule

Can you see how to make a 3:8 or 4:16 decoder in the same fashion?
Multi-bit Design and Parameterization
A Dataflow MUX - Multi-Bit

module mux21(q, sel, a, b);
    input sel;
    input[15:0] a, b;
    output[15:0] q;

    assign q = sel ? b : a;
endmodule

Key Ideas:
The predicate must evaluate to true or false (1 or 0)
The parts getting assigned must be all same widths.
A Dataflow MUX - Parameterized Width

```verilog
module mux21n(q, sel, a, b);
  parameter WID = 16;
  input sel;
  input[WID-1:0] a, b;
  output[WID-1:0] q;
  assign q = sel ? b : a;
endmodule

• By default, this is now a 16-bit wide MUX.
• When instantiating, the default value of 16 can be overridden:

  mux21n M1(q, sel, a, b);         // Instance a 16-bit version
  mux21n #(4) M0(q, sel, a, b);    // Instance a 4-bit version

Does this work for a 1-bit MUX?
Using Parameterization

• Careful planning often allows you to write one design which can be reused
  - *Reuse* is a common goal in design
    • *Simplifies your work*
    • *Eliminates errors*
    • *Saves time later*

• Whenever possible, plan for reuse
Parameterization Exercise

• Design a 4:1 MUX that works with any size operands (arbitrary bit-width)

• Either:
  - Build arbitrary bit-width 2:1 MUX
  - Structurally instance 3 of these to make a 4:1 MUX
    or
  - Write an arbitrary bit-width 4:1 MUX using the ?: operator
4:1 MUX - Method 1

module mux41n(q, sel, a, b, c, d);
    parameter WID=16;
    input[1:0] sel;
    input[WID-1:0] a, b, c, d;
    output[WID-1:0] q;
    wire[WID-1:0] tmp1, tmp2;

    mux21n #(WID) M0(tmp1, sel[0], a, b);
    mux21n #(WID) M1(tmp2, sel[0], c, d);
    mux21n #(WID) M2(q, sel[1], tmp1, tmp2);

endmodule

If the mux21n cells are parameterizable for bit-width this works...
If not, it doesn't work...
4:1 MUX - Method 2

module mux41(q, sel, a, b, c, d);
    parameter WID=16;
    input[1:0] sel;
    input[WID-1:0] a, b, c, d;
    output[WID-1:0] q;

    assign q = (sel==2'b00) ? a:
        (sel==1)     ? b:
        (sel==2'b10) ? c:
            d;

endmodule

Cascaded ?: operators form an if-then-else structure

Note how sel can be compared to bit patterns (2'b00) or to numbers (1)
Behavioral Verilog
Used for Sequential Circuits and Combinational Circuits

```verilog
module dff(clk, d, q);
    input clk, d;
    output reg q;

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

You can use this as a DFF for your design.

Figure out how to parameterize it for arbitrary input/output widths

Remainder of behavioral design not covered in this class...
Conclusion

• With what you know...
  - You can do reasonable designs
  - Must structurally instance all storage elements (flip flops)

• Behavioral Design
  - A number of nuances with respect to timing semantics
  - Not recommended beyond simple FF’s for this class
    • You will learn full range of behavioral design in later courses