Announcements

- Lab 1 due date postponed
  - Due: Wed. Oct 8 @ 6PM (via e-mail)
- Lab 2, Part 1 will be up early next week
  - Design fetch unit for your processor
    - Look over description before next lecture!
- Stay up-to-date with webboard
  - Sign up for mailing lists so you get posts automatically
A simple example (comb. circuit)

- Let's design a 1 bit full adder (RTL style)

```
module FA(input a, b, cin, output s, cout);
    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

- Ok, but what if we want more than 1 bit FA?

Adapted from Arvind & Asanovic’s MIT 6.375 lecture

---

A simple D flip flop (seq. circuit)

- For sequential circuits, use always blocks
- Always blocks (and assign) are executed in parallel!

```
module DFF(input clk, d, output q, q_bar);
    reg q, q_bar;
    always @(posedge clk) // triggered on the rising edge of the clock
    begin
        q <= d; // non-blocking assignment (LHS not updated until later)
        q_bar <= ~d;
        // q_bar <= ~q will not function correctly! Why not? */
    end
endmodule
```

Adapted from Arvind & Asanovic’s MIT 6.375 lecture
Always blocks in comb. circuits

- Can use continual assignment OR always blocks for combinational circuits
- Our 1-bit adder using always block

```verbatim
define module FA(
    input a, b, cin,
    output s, cout)

reg s, cout; // when using always block, LHS must be reg type
always @ (*) // for comb circuits, use "*" so avoid errors
begin
    s = a ^ b ^ cin; // use blocking assignment here (LHS immediately)
    cout = (a & b) | (a & cin) | (b & cin);
end
endmodule
```

Blocking vs. non-blocking assignment

- Order of blocking statements matter
  - These are not the same
    - `c = a + b;`
    - `d = c + e;`
    - `d = c + e;`
    - `c = a + b;`
- Order of non-blocking statements doesn’t
  - These are the same
    - `c <= a + b;`
    - `d <= c + e;`
    - `d <= c + e;`
    - `c <= a + b;`
- Use non-blocking with sequential, blocking with combinational
Blocking vs. Non-blocking assignments

- Order of blocking statements matters
- Assume $a = 1$, $b = 2$, $c = 4$, $d = 4$, $e = 5$ initially
  
  ```
  // ex1
  c = a + b;
  d = c + e;
  d = c + e;
  c = a + b;
  // ex2
  c = a + b;
  d = c + e;
  d = c + e;
  c = a + b;
  ```

- Ex1: $c = 3$, $d = 8$
- Ex2: $c = 3$, $d = 9$

- Order of non-blocking statements doesn’t
- Example: (same assumptions)
  
  ```
  // ex1
  c <= a + b;
  d <= c + e;
  d <= c + e;
  c <= a + b;
  // ex2
  c <= a + b;
  d <= c + e;
  d <= c + e;
  c <= a + b;
  ```

- Ex1: $c = 3$, $d = 9$
- Ex2: $c = 3$, $d = 9$

Guideline: Use blocking for combination, non-blocking for sequential

Verilog Simulation Behavior

- Always blocks and “assign” statements execute in parallel
  - Signals in sensitivity list (“@(...)”) trigger always blocks
  - “Assign” trigged when RHS signal changes
- Non-blocking assignment is 2-step processes
  - Step 1: Evaluate RHS (beginning of time step)
  - Step 2: Update LHS (end of time step)
**Tips for maintaining synthesizability**

- Only leaf modules should have functionality
  - All other modules are strictly structural, i.e., they only wire together sub-modules
- Use only positive-edge triggered flip-flops for state
- Do not assign to the same variable from more than one always block
- Separate combinational logic from sequential logic
- Avoid loops like the plague
  - Use for and while loops only for test benches

Adapted from Arvind & Asanovic's MIT 6.375 lecture

---

**Data Types in Verilog**

- Basic type: bit vector
  - Values: 0, 1, X (don't care), Z (high impedance)
- Bit vectors expressed in multiple ways:
  - binary: 4'b11_10 ( _ is just for readability)
  - hex: 16'h034f
  - decimal: 32'd270
  - other formats but these are the most useful
Data types (continued)

- Connect things together with wires
  - Single wire:
    - `wire my_wire;`
  - “Array” of wires
    - `wire[7:0] my_wire;`
      - Why not `wire[0:7]?`
- For always blocks, must use “reg” type
  - Single reg or an array of reg
    - `reg[3:0] accum; // 4 bit “reg”`
  - `reg` is not necessarily a hardware register

Verilog RTL Operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>%</th>
<th>**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>!</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;=</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>!=</td>
<td>===</td>
<td>!==</td>
<td>===</td>
<td>!==</td>
</tr>
<tr>
<td>Bitwise</td>
<td>~</td>
<td>&amp;</td>
<td></td>
<td></td>
<td>^</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduction</th>
<th>&amp; &amp;</th>
<th>~</th>
<th>~</th>
<th>^</th>
<th>^ ~</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;&gt;</td>
<td>&lt;&lt;&lt;</td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td>{</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>?</td>
<td>;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Avoid using %, **, and / because you'll run into problems when trying to synthesis

Adapted from Arvind & Asanovic’s MIT 6.375 lecture
Mystery Combinational Circuit

- We can use case statements within an always block
- What hardware will this turn into?

```verilog
module mystery( 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

Adapted from Arvind & Asanovic’s MIT 6.375 lecture
```

Finite State Machines (FSMs)

- Useful for designing many different types of circuits
- 3 basic components:
  - Combinational logic (next state)
  - Sequential logic (store state)
  - Output logic
- Different encodings for state:
  - Binary (min FF’s), Gray, One hot (good for FPGA), One cold, etc
A simple FSM in Verilog

```
module simple_fsm( input clk, start, 
                  output restart);

  reg [1:0] state, next_state;
  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10; // binary encode

  always @ (*)
    begin : next_state_logic
      case ( state )
        S0: begin
          if ( start )
            next_state = S1;
          else
            next_state = S0;
        end
        S1: begin
          next_state = S2;
        end
        S2: begin
          if (restart)
            next_state = S0;
          else
            next_state = S2;
        end
        default: next_state = S0;
      endcase
    end

  // continued to the right

  always @ (posedge clk)
    begin : state_assignment
      state <= next_state;
    end

endmodule
```

Tips on FSMs

- Don’t forget to handle the default case
- Use two different always blocks for next state and state assignment
  - Can do it in one big block but not as clear
- Outputs can be a mix of combin. and seq.
  - Moore Machine: Output only depends on state
  - Mealy Machine: Output depends on state and inputs
Next step: Design your own HW

- You now have the basics of Verilog
  - Can create simple circuits
    - Combinational, Sequential, or Mixed
  - Beware of teh Internetz
    - A lot of code out there isn’t synthesizable
- How do the real HW designers code?
  - Next: More advanced design techniques