Review

• Verilog is the most-commonly used HDL
• We have seen combinatorial constructs
  • Assign statement
  • Always blocks

• Practice is the best way to learn a new language…
Latches and Flip-Flops

- **Flip-flop is edge-triggered, latch is level-sensitive**

- **D Flip-flop**
  - Rising edge

- **D Latch**
  - Transparent
    - **HIGH**
      - Level sensitive if there is no ‘edge’
Timing

• Combinational logic timing

A is arriving late (is in the critical path)

B is arriving late (is in the critical path)

HL and LH transition differ
\( t_{A\text{-Out}} \) and \( t_{B\text{-Out}} \) differ

In CMOS, propagation delay depends on:
- Gate type, size (output resistance)
- Capacitive loading
- Input slope

Timing

• Flip-flop timing

(latch timing will be covered later)

• Setup and hold times

Data cannot change in the interval of setup time before the clock edge to hold time after the clock edge
Register

• 4-bit register

Accumulator

Administrivia

• Homework 2 is due this Friday
  • Homework 3 will be posted this week
  • Lab 3 this week
State Elements in Verilog

Always blocks are the only way to specify the “behavior” of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

```verilog
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;
  always @(posedge clk)
    if (rst)
       q <= 1'b0;
    else if (set)
       q <= 1'b1;
    else
       q <= d;
  endmodule
```

Unlike logic gates, there are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or standard-cell specific flip-flops.
The Sequential *always* Block

**Combinational**

```
module comb(input a, b, sel, output reg out);
    always @(*) begin
        if (sel) out = b;
        else out = a;
    end
endmodule
```

**Sequential**

```
module seq(input a, b, sel, clk, output reg out);
    always @(posedge clk) begin
        if (sel) out <= b;
        else out <= a;
    end
endmodule
```

---

### Latches vs. Flip-Flops

**Flip-Flop**

```
module flipflop
(
    input clk,
    input d,
    output reg q
);
    always @(posedge clk)
    begin
        q <= d;
    end
endmodule
```

**Latch**

```
module latch
(
    input clk,
    input d,
    output reg q
);
    always @(clk or d)
    begin
        if (clk) q <= d;
    end
endmodule
```
Importance of the Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)

Note: The following is incorrect syntax: `always @(clear or negedge clock)`
If one signal in the sensitivity list uses `posedge`/`negedge`, then all signals must.

- Assign any signal or variable from only one `always` block.

Be wary of race conditions: `always` blocks with same trigger execute concurrently...

### Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.

  - **Blocking assignment (=):** evaluation and assignment are immediate

    ```verilog
    always @(*) begin
        x = a | b;  // 1. evaluate a|b, assign result to x
        y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
        z = b & ~c;  // 3. evaluate b&(~c), assign result to z
    end
    ```

  - **Nonblocking assignment (<=):** all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active `always` blocks)

    ```verilog
    always @(*) begin
        x <= a | b;  // 1. evaluate a|b, but defer assignment to x
        y <= a ^ b ^ c;  // 2. evaluate a^b^c, but defer assignment to y
        z <= b & ~c;  // 3. evaluate b&(~c), but defer assignment to z
        // 4. end of time step: assign new values to x, y and z
    end
    ```

Sometimes, as above, both produce the same result. Sometimes, not!
Assignment Styles for Sequential Logic

What we want:
Register-based digital delay line (a.k.a. shift-register)

Will non-blocking and blocking assignments both produce the desired result?

```verilog
module nonblocking(  
  input in, clk,  
  output reg out  
);  
  reg q1, q2;  
  always @(posedge clk) begin  
    q1 <= in;  
    q2 <= q1;  
    out <= q2;  
  end  
endmodule
```

```verilog
module blocking(  
  input in, clk,  
  output reg out  
);  
  reg q1, q2;  
  always @(posedge clk) begin  
    q1 = in;  
    q2 = q1;  
    out = q2;  
  end  
endmodule
```

What we want:
Register-based digital delay line (a.k.a. shift-register)

Use Nonblocking for Sequential Logic

```verilog
always @(posedge clk) begin  
  q1 <= in;  
  q2 <= q1;  // uses old q1  
  out <= q2;  // uses old q2  
end
```

```verilog
always @(posedge clk) begin  
  q1 = in;  
  q2 = q1;  // uses new q1  
  out = q2;  // uses new q2  
end
```

("old" means value before clock edge, "new" means the value after most recent assignment)

“At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2.”

“At each rising clock edge, q1 = in. After that, q2 = q1. After that, out = q2. Therefore out = in.”

- Blocking assignments **do not** reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use **nonblocking** assignments for sequential **always** blocks
Example: A Simple Counter

// 4-bit counter with enable and synchronous clear
module counter(input clk, enb, clr, output reg [3:0] count);
  always @(posedge clk) begin
    count <= clr ? 4'b0 : (enb ? count+1 : count);
  end
endmodule

Example - Parallel to Serial Converter

module ParToSer(ld, X, out, clk);
  input [3:0] X;
  input ld, clk;
  output out;
  reg [3:0] Q;
  wire [3:0] NS;
  assign NS = ld ? X : {Q[0], Q[3:1]};
  always @(posedge clk)
    Q <= NS;
  assign out = Q[0];
endmodule
Simplified Verilog Guidelines

• Combinational logic:
  • Continuous Assignment:
    ```verilog
    assign a = b & c;
    ```
  • Always block with @(*)
    ```verilog
    always @(*) begin
    a = b & c;  // blocking statement
    end
    ```

• Sequential logic:
  • Always block with @(posedge clk)
    ```verilog
    always @(posedge clk) begin
    a <= b & c;  // nonblocking statement
    end
    ```

Verilog in EECS 151/251A

• We use behavioral modeling at the bottom of the hierarchy
• Use instantiation to 1) build hierarchy and,
  2) map to FPGA and ASIC resources not supported by synthesis.
• Favor continuous assign and avoid always blocks unless:
  • No other alternative: ex: state elements, case
  • Helps readability and clarity of code: ex: large nested if else
• Use named ports.
• Verilog is a big language. This is only an introduction.
  • Harris & Harris book chapter 4 is a good source.
  • Be careful of what you read on the web. Many bad examples out there.
  • We will be introducing more useful constructs throughout the semester. Stay tuned!
Verilog vs. SystemVerilog

• **always** statements in Verilog can be used to infer flip-flops, latches or logic
  • Depends on the sensitivity list and the statement
  • Easy to create confusion

• System Verilog adds disambiguation:
  • **always_ff** for flip-flops
  • **always_latch** for latches
  • **always_comb** for combinational logic

Verilog Testbenches
Simulating the Circuit

• Once you have a circuit in Verilog (device under test, or DUT), you would like to test it

• Instantiate the DUT and supply its inputs via a testbench
  • Simple
  • Comprehensive
  • Random

• `initial` statement supplies the stimuli

Testbench basics

• Example clock
  ```verilog
  reg clk;

  initial clk = 0;
  always #(CLOCK_PERIOD/2) clk <= ~clk;
  ```

• Example inputs
  ```verilog
  initial begin
  in <= 4’h0;
  @(negedge clk) in<= 4’h1;
  ...
  (sets up inputs on the negedge, so they are ready at the posedge)
  ```

Only small DUTs can be tested exhaustively
SystemVerilog

- SystemVerilog adds many more verification features
  - We will touch on assertions and covers
    (relates to CS70)

Final Thoughts on Verilog Examples

Verilog looks like C, but it describes hardware:
Entirely different semantics: multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

As you get more practice, you will know how to best write Verilog for a desired result.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.
Clicker Question

• How many stimuli to exhaustively test a 32-b adder?
  A) 32
  B) 64
  C) 65,536
  D) 4,294,967,296
  E) 18,446,744,073,709,551,616

www.yellkey.com/perform

Combinational Logic
**Combinational Logic**

- The outputs depend *only* on the current values of the inputs.
  - Memoryless: compute the output values using the current inputs.

![Combinational Logic Diagram](image)

**Combinational Logic Example**

**Truth Table Description:**

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Boolean Equations:**

\[ y = x_0 \text{ OR } x_1 \]
\[ = x_0 + x_1 \]

**Gate Representations:**

![Gate Diagram](image)
Relationship Among Representations

Truth Table

Boolean Expression

Gate Representation

Unique

Convenient for manipulation

Close to Implementation

Boolean Algebra
Boolean Algebra Background

- Logic: The study of the principles of reasoning.
- The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
  - His variables took on TRUE, FALSE.
- Later Claude Shannon (father of information theory) showed (in his Master’s thesis!) how to map Boolean Algebra to digital circuits.

Boolean Algebra Fundamentals

- Two elements \{0, 1\}
- Two binary operators: AND (\cdot\) OR (+)
- One unary operator: NOT (\bar{\cdot}, \bar{)}
Boolean Operations

- Given two variables \((x, y)\), 16 logic functions

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(F_0)</th>
<th>(F_1)</th>
<th>(F_2)</th>
<th>(F_3)</th>
<th>(F_4)</th>
<th>(F_5)</th>
<th>(F_6)</th>
<th>(F_7)</th>
<th>(F_8)</th>
<th>(F_9)</th>
<th>(F_A)</th>
<th>(F_B)</th>
<th>(F_C)</th>
<th>(F_D)</th>
<th>(F_E)</th>
<th>(F_F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Laws of Boolean Algebra

- Identities, null elements:
  - \(x + 0 = x, x \cdot 1 = x\)
  - \(x + 1 = 1, x \cdot 0 = 0\)

- Idempotency:
  - \(x + x = x, x \cdot x = x\)

- Complements:
  - \(x + x' = 1, x \cdot x' = 0\)

- Commutativity:
  - \(x + y = y + x, x \cdot y = y \cdot x\)

- Associativity:
  - \((x + y) + z = x + (y + z) = x + y + z\)
  - \((x \cdot y) \cdot z = x \cdot (y \cdot z) = x \cdot y \cdot z\)

- Distributivity:
  - \(x \cdot (y + z) = (x \cdot y) + (x \cdot z)\)
  - \(x + (y \cdot z) = (x + y) \cdot (x + z)\)

- Duality:
  - \(\text{AND} \rightarrow \text{OR} \text{ and vice versa}\)
  - \(0 \rightarrow 1 \text{ and vice versa}\)
  - Leave literals unchanged

\[
\{F(x_1, x_2, \ldots, x_n, 0, 1, +, *)\}^D = \{F(x_1, x_2, \ldots, x_n, 1, 0, +, *)\}
\]

Literals are variables or their complements
## Proving Distributive Law

\[ X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(Y + Z)</th>
<th>X \cdot (Y + Z)</th>
<th>(X \cdot Y)</th>
<th>(X \cdot Z)</th>
<th>(X \cdot Y) + (X \cdot Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Proving Distributive Law

\[ X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(Y + Z)</th>
<th>X \cdot (Y + Z)</th>
<th>(X \cdot Y)</th>
<th>(X \cdot Z)</th>
<th>(X \cdot Y) + (X \cdot Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DeMorgan's Law

- Theorem for complementing a complex function.

\[(x + y)' = x' y'\]

\[(x y)' = x' + y'\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>(x + y)'</th>
<th>x' y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>(x y)'</th>
<th>x' + y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DeMorgan's Law

- Procedure for complementing a complex function.

\[(x + y)' = x' y'\]

\[(x y)' = x' + y'\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>(x + y)'</th>
<th>x' y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x'</th>
<th>y'</th>
<th>(x y)'</th>
<th>x' + y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Summary

• Sequential logic uses flip-flops and (sometimes) latches
• Flip-flops and latches are inferred in Verilog
  • Always blocks
  • Practice is the best way to learn a new language...
• Blocking and non-blocking assignments
• Combinational logic block outputs depend only on its inputs
• Boolean algebra can be used for manipulation and simplification of Boolean equations