Required Readings (This Week)

- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full

- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By tomorrow, make sure you are done with
  - P&P Chapters 1-3 + H&H Chapters 1-4
Required Readings (Next Week)

- Von Neumann Model, LC-3, and MIPS
  - P&P, Chapter 4, 5
  - H&H, Chapter 6
  - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
  - H&H, Appendix B (MIPS instructions)

- Programming
  - P&P, Chapter 6

- Recommended: Digital Building Blocks
  - H&H, Chapter 5
Agenda

- Hardware Description Languages
- Implementing Combinational Logic (in Verilog)
- Implementing Sequential Logic (in Verilog)

- The Verilog slides constitute a tutorial. We will not cover all.
- All slides will be beneficial for your labs.
Hardware Description Languages
& Verilog
2017: Intel Kaby Lake

- 64-bit processor
- 4 cores, 8 threads
- 14-19 stage pipeline
- 3.9 GHz clock
- 1.75B transistors
- In ~47 years, about 1,000,000-fold growth in transistor count and performance!

How to Deal with This Complexity?

- **Hardware Description Languages!**

- A fact of life in computer engineering
  - Need to be able to **specify complex designs**
    - communicate with others in your design group
  - ... and to **simulate** their behavior
    - *yes, it’s what I want to build*
  - ... and to **synthesize** (automatically design) portions of it
    - have an error-free path to implementation

- **Hardware Description Languages**
  - Many similarly featured **HDLs** (e.g., **Verilog**, **VHDL**, ...) 
    - if you learn one, it is **not hard to learn** another
    - mapping between languages is typically **mechanical**, especially for the commonly used subset
Two well-known hardware description languages

- **Verilog**
  - Developed in 1984 by Gateway Design Automation
  - Became an IEEE standard (1364) in 1995
  - More popular in US

- **VHDL (VHSIC Hardware Description Language)**
  - Developed in 1981 by the US Department of Defense
  - Became an IEEE standard (1076) in 1987
  - More popular in Europe

In this course we will use Verilog
Hardware Design Using Verilog
Hierarchical Design

- **Design hierarchy of modules is built using instantiation**
  - Predefined “primitive” gates (AND, OR, ...)
  - Simple modules are built by instantiating these gates (components like MUXes)
  - Other modules are built by instantiating simple components, ...

- **Hierarchy controls complexity**
  - Analogous to the use of function abstraction in SW

- **Complexity is a BIG deal**
  - In real world how big is size of one “blob” of random logic that we would describe as an HDL, then synthesize to gates?

[https://techreport.com/review/21987/intel-core-i7-3960x-processor](https://techreport.com/review/21987/intel-core-i7-3960x-processor)
Top-Down Design Methodology

- We define the top-level module and identify the sub-modules necessary to build the top-level module.
- Subdivide the sub-modules until we come to leaf cells.
  - Leaf cell: circuit components that cannot further be divided (e.g., logic gates, cell libraries)
Bottom-Up Design Methodology

- We first identify the **building blocks** that are available to us
- **Build bigger modules**, using these building blocks
- These modules are then used for higher-level modules until we build the **top-level module** in the design
A module is the main building block in Verilog

We first need to define:
- Name of the module
- Directions of its ports (e.g., input, output)
- Names of its ports

Then:
- Describe the functionality of the module
Implementing a Module in Verilog

module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

    // here comes the circuit description
endmodule
The following two codes are functionally identical:

```verilog
module test ( a, b, y );
  input a;
  input b;
  output y;
endmodule
```

```verilog
module test ( input a, input b, output y );
endmodule
```

Port name and direction declaration can be combined.
What If We Have Multi-bit Input/Output?

- You can also define multi-bit Input/Output (Bus)
  - [range_end : range_start]
  - **Number of bits:** range_end – range_start + 1

- Example:

```vhdl
input [31:0] a; // a[31], a[30] .. a[0]
output [15:8] b1; // b1[15], b1[14] .. b1[8]
output [7:0] b2; // b2[7], b2[6] .. b2[0]
input c; // single signal
```

- **a** represents a 32-bit value, so we prefer to define it as:
  - [31:0] a

- It is preferred over [0:31] a which resembles *array* definition

- It is good practice to be **consistent** with the representation of multi-bit signals, i.e., always [31:0] or always [0:31]
Manipulating Bits

- Bit Slicing
- Concatenation
- Duplication
Basic Syntax

- Verilog is case sensitive
  - `SomeName` and `somename` are not the same!
- Names cannot start with numbers:
  - `2good` is not a valid name
- Whitespaces are ignored

```verbatim
// Single line comments start with a //

/* Multiline comments are defined like this */
```
Two Main Styles of HDL Implementation

- **Structural (Gate-Level)**
  - The module body contains *gate-level description* of the circuit
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - ... and interconnections between these modules
  - Describes a hierarchy

- **Behavioral**
  - The module body contains *functional description* of the circuit
  - Contains logical and mathematical *operators*
  - **Level of abstraction is higher than gate-level**
    - Many possible gate-level realizations of a behavioral description

- **Practical circuits use a combination of both**
Structural (Gate-Level) HDL
Schematic of module “top” that is built from two instances of module “small”
Module Definitions in Verilog

module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;

// description of small
endmodule
Structural HDL Example

- **Defining wires (module interconnections)**

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;
    // description of small
endmodule
```
The first instantiation of the “small” module

```verilog
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;
  // instantiate small once
  small i_first ( .A(A),
                .B(SEL),
                .Y(n1)   );
endmodule

module small (A, B, Y);
  input A;
  input B;
  output Y;
  // description of small
endmodule
```
The second instantiation of the “small” module

module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // instantiate small once
    small i_first ( .A(A),
                    .B(SEL),
                    .Y(n1)   );

    // instantiate small second time
    small i_second ( .A(n1),
                     .B(C),
                     .Y(Y) );

endmodule

module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small
    endmodule
### Short form of module instantiation

**Module top**:

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // alternative
    small i_first ( A, SEL, n1 );

    /* Shorter instantiation, pin order very important */
    // any pin order, safer choice
    small i_second ( .B(C),
                    .Y(Y),
                    .A(n1) );
endmodule
```

**Module small**:

```verilog
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small
endmodule
```

**Diagram**:

- `i_first` module with inputs A, SEL, and n1.
- `i_second` module with inputs B, Y, and A(n1).
- `top` module with inputs A, SEL, and C, output Y.

---

**Short form is not good practice as it reduces code maintainability**
Verilog supports basic logic gates as predefined *primitives*

- These primitives are *instantiated* like modules except that they are predefined in Verilog and *do not need a module definition*.

```verilog
module mux2(input d0, d1,
            input s,
            output y);
  wire ns, y1, y2;
  not g1 (ns, s);
  and g2 (y1, d0, ns);
  and g3 (y2, d1, s);
  or  g4 (y, y1, y2);
endmodule
```
Behavioral HDL
Recall: Two Main Styles of HDL Implementation

- **Structural (Gate-Level)**
  - The module body contains *gate-level description* of the circuit
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - ... and interconnections between these modules
  - Describes a hierarchy

- **Behavioral**
  - The module body contains *functional description* of the circuit
  - Contains logical and mathematical operators
  - **Level of abstraction is higher than gate-level**
    - Many possible gate-level realizations of a behavioral description

- Practical circuits would use a combination of both
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

    // here comes the circuit description
    assign y = ~a & ~b & ~c |
               a & ~b & ~c |
               a & ~b & c;

endmodule
A behavioral implementation still models a hardware circuit!
module gates(input [3:0] a, b,
       output [3:0] y1, y2, y3, y4, y5);

/* Five different two-input logic gates acting on 4 bit buses */

assign y1 = a & b;  // AND
assign y2 = a | b;  // OR
assign y3 = a ^ b;  // XOR
assign y4 = ~(a & b);  // NAND
assign y5 = ~(a | b);  // NOR
endmodule
Bitwise Operators: Schematic View
module and8(input [7:0] a,  
    output y);

assign y = &a;

// &a is much easier to write than  

endmodule
Reduction Operators: Schematic View

8-input AND gate
Conditional Assignment in Behavioral Verilog

module mux2(input [3:0] d0, d1,
             input    s,
             output [3:0] y);

    assign y = s ? d1 : d0;
    // if (s) then y=d1 else y=d0;
endmodule

- ? : is also called a **ternary operator** as it operates on three inputs:
  - s
  - d1
  - d0
Conditional Assignment: Schematic View
module mux4(input [3:0] d0, d1, d2, d3
    input [1:0] s,
    output [3:0] y);

    assign y = s[1] ? ( s[0] ? d3 : d2)
        : ( s[0] ? d1 : d0);

    // if (s1) then
    //    if (s0) then y=d3 else y=d2
    // else
    //    if (s0) then y=d1 else y=d0

endmodule
Even More Complex Conditional Assignments

```verbatim
module mux4(input [3:0] d0, d1, d2, d3
            input [1:0] s,
            output [3:0] y);

    assign y = (s == 2’b11) ? d3 :
                 (s == 2’b10) ? d2 :
                 (s == 2’b01) ? d1 :
                          d0;

    // if      (s = "11")  then y= d3
    // else if (s = "10")  then y= d2
    // else if (s = "01")  then y= d1
    // else                     y= d0

endmodule
```

# Precedence of Operations in Verilog

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
</tr>
<tr>
<td>*, /, %</td>
<td>mult, div, mod</td>
</tr>
<tr>
<td>+, -</td>
<td>add, sub</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>shift</td>
</tr>
<tr>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td>arithmetic shift</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>comparison</td>
</tr>
<tr>
<td>==, !=</td>
<td>equal, not equal</td>
</tr>
<tr>
<td>&amp;, &amp;&amp;</td>
<td>AND, NAND</td>
</tr>
<tr>
<td>^, ^^</td>
<td>XOR, XNOR</td>
</tr>
<tr>
<td></td>
<td>, ~</td>
</tr>
<tr>
<td>?:</td>
<td>ternary operator</td>
</tr>
</tbody>
</table>
How to Express Numbers?

N’ Bxx
8’ b0000_0001

- **(N) Number of bits**
  - Expresses how many bits will be used to store the value

- **(B) Base**
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)

- **(xx) Number**
  - The value expressed in base
  - Can also have X (invalid) and Z (floating), as values
  - Underscore _ can be used to improve readability
## Number Representation in Verilog

<table>
<thead>
<tr>
<th>Verilog</th>
<th>Stored Number</th>
<th>Verilog</th>
<th>Stored Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4'b1001</td>
<td>1001</td>
<td>4'd5</td>
<td>0101</td>
</tr>
<tr>
<td>8'b1001</td>
<td>0000 1001</td>
<td>12'hFA3</td>
<td>1111 1010 0011</td>
</tr>
<tr>
<td>8'b0000_1001</td>
<td>0000 1001</td>
<td>8'o12</td>
<td>00 001 010</td>
</tr>
<tr>
<td>8'bxX0X1zZ1</td>
<td>XXOX 1ZZ1</td>
<td>4'h7</td>
<td>0111</td>
</tr>
<tr>
<td>'b01</td>
<td>0000 .. 0001</td>
<td>12'h0</td>
<td>0000 0000 0000</td>
</tr>
</tbody>
</table>

32 bits (default)
Floating signal: Signal that is not driven by any circuit
- Open circuit, floating wire

Also known as: high impedance, hi-Z, tri-stated signals

```verilog
module tristate_buffer(input [3:0] a,
                        input en,
                        output [3:0] y);

  assign y = en ? a : 4'bz;

endmodule
```

Diagram:
```
  en
  +--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+        ```
Recall: Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

Figure 2.40 Tristate buffer
Recall: Example Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
  - At any time only the CPU or the memory can place a value on the wire, both not both
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time
Recall: Example Design with Tri-State Buffers
Truth Table for AND with Z and X

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>AND</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
What Happens with HDL Code?

- **Synthesis**
  - Modern tools are able to map synthesizable HDL code into low-level cell libraries → netlist describing gates and wires.
  - They can perform many optimizations.
  - ... however they **can not** guarantee that a solution is optimal.
    - Mainly due to computationally expensive placement and routing algorithms.
  - Most common way of Digital Design these days.

- **Simulation**
  - Allows the behavior of the circuit to be **verified without actually manufacturing** the circuit.
  - Simulators can work on structural or behavioral HDL.
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

    // here comes the circuit description
    assign y = ~a & ~b & ~c |
       a & ~b & ~c |
       a & ~b & c;

endmodule
Synthesizing the “example”

\[
\text{AND} \quad \text{AND} \quad \text{AND} \quad \text{OR}
\]

\[
a \rightarrow \text{AND} \quad b \rightarrow \text{AND} \quad c \rightarrow \text{AND} \quad \rightarrow \text{OR} \quad \rightarrow \ y
\]
Simulating the “example”

Waveform Diagram

<table>
<thead>
<tr>
<th>Now: 800 ns</th>
<th>0 ns</th>
<th>160</th>
<th>320 ns</th>
<th>480</th>
<th>640 ns</th>
<th>800 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What We Have Seen So Far

- Describing structural hierarchy with Verilog
  - Instantiate modules in another module
- Describing functionality using behavioral modeling
- Writing simple logic equations
  - We can write AND, OR, XOR, ...
- Multiplexer functionality
  - If ... then ... else
- We can describe constants
- But there is more...
More Verilog Examples

- We can write Verilog code in **many different ways**
- Let’s see how we can express the same functionality by developing Verilog code
  - At **low-level**
    - Poor readability
    - More optimization opportunities (especially for low-level tools)
  - At a **higher-level of abstraction**
    - Better readability
    - Limited optimization opportunities
Comparing Two Numbers

- Defining your own gates as new modules

- We will use our gates to show the different ways of implementing a 4-bit comparator (equality checker)

An XNOR gate

```verilog
module MyXnor (input A, B, output Z);
    assign Z = ~(A ^ B); //not XOR
endmodule
```

An AND gate

```verilog
module MyAnd (input A, B, output Z);
    assign Z = A & B; // AND
endmodule
```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
    output eq);
    wire c0, c1, c2, c3, c01, c23;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
Using Logical Operators

```verilog
module compare (input a0, a1, a2, a3, b0, b1, b2, b3, output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0));  // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1));  // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2));  // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3));  // XNOR

    assign c01 = c0 & c1;
    assign c23 = c2 & c3;
    assign eq  = c01 & c23;

endmodule
```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3, output eq);
    wire c0, c1, c2, c3;
    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    // assign c01 = c0 & c1;
    // assign c23 = c2 & c3;
    // assign eq = c01 & c23;
    assign eq = c0 & c1 & c2 & c3;
endmodule
module compare (input [3:0] a, input [3:0] b, output eq);
wire [3:0] c; // bus definition

MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0])); // XNOR
MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1])); // XNOR
MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2])); // XNOR
MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3])); // XNOR

assign eq = &c; // short format

endmodule
Bitwise Operations

module compare (input [3:0] a, input [3:0] b, output eq);
    wire [3:0] c; // bus definition

    // MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]));
    // MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]));
    // MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]));
    // MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]));

    assign c = ~(a ^ b); // XNOR

    assign eq = &c; // short format

endmodule
module compare (input [3:0] a, input [3:0] b, output eq);

// assign c = ~(a ^ b); // XNOR

// assign eq = &c; // short format

assign eq = (a == b) ? 1 : 0; // really short

endmodule
Writing More Reusable Verilog Code

- We have a module that can compare two 4-bit numbers

- What if in the overall design we need to compare:
  - 5-bit numbers?
  - 6-bit numbers?
  - ...
  - N-bit numbers?
  - Writing code for each case looks tedious

- What could be a better way?
Parameterized Modules

In Verilog, we can define module parameters

```verilog
module mux2
    #(parameter width = 8) // name and default value
    (input [width-1:0] d0, d1,
     input s,
     output [width-1:0] y);

    assign y = s ? d1 : d0;
endmodule
```

We can set the parameters to different values when instantiating the module
module mux2
  #(parameter width = 8) // name and default value
  (input [width-1:0] d0, d1,
   input s,
   output [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
What About Timing?

- It is possible to define *timing relations* in Verilog. **BUT:**
  - These are **ONLY** for simulation
  - They **CAN NOT** be synthesized
  - They are used for *modeling delays* in a circuit

```verilog
//timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a;  // inverted output after 5ns
assign #9 z2 = a;   // output after 9ns

endmodule
```

More to come later today!
Good Practices

- Develop/use a **consistent** naming style

- Use **MSB to LSB ordering** for buses
  - Use “a[31:0]”, **not** “a[0:31]”

- Define **one module per file**
  - Makes managing your design hierarchy easier

- Use a file name that equals module name
  - e.g., module **TryThis** is defined in a file called **TryThis.v**

- Always keep in mind that **Verilog describes hardware**
We have seen an overview of Verilog

Discussed structural and behavioral modeling

Showed combinational logic constructs
Implementing Sequential Logic Using Verilog
Combinational + Memory = Sequential
Sequential Logic in Verilog

- Define blocks that have memory
  - *Flip-Flops, Latches, Finite State Machines*

- **Sequential Logic** state transition is triggered by a “CLOCK” signal
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of signal

- Combinational constructs are **not** sufficient
  - We need **new constructs**:
    - *always*
    - *posedge/negedge*
The “always” Block

```verilog
always @ (sensitivity list)
  statement;
```

Whenever the event in the sensitivity list occurs, the statement is executed.
Example: D Flip-Flop

```verilog
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

always @(posedge clk)
  q <= d;                  // pronounced "q gets d"

endmodule
```

- **posedge** defines a rising edge (transition from 0 to 1).
- Statement executed when the **clk signal rises (posedge of clk)**
- Once the clk signal rises: the value of **d** is copied to **q**
Example: D Flip-Flop

```verilog
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

always @(posedge clk)
  q <= d; // pronounced “q gets d”

endmodule
```

- **assign** statement is **not** used within an always block
- `<=` describes a **non-blocking** assignment
  - We will see the difference between **blocking assignment** and **non-blocking** assignment soon
Example: D Flip-Flop

module flop(input clk,
             input [3:0] d,
             output reg [3:0] q);

always @(posedge clk)
  q <= d;        // pronounced “q gets d”

endmodule

- Assigned variables need to be declared as `reg`
- The name `reg` does not necessarily mean that the value is a register (It could be, but it does not have to be)
- We will see examples later
Asynchronous and Synchronous Reset

- **Reset** signals are used to **initialize** the hardware to a known state
  - Usually activated **at system start** (on power up)

- **Asynchronous Reset**
  - The reset signal is sampled **independent** of the clock
  - Reset gets the highest priority
  - Sensitive to **glitches**, may have **metastability** issues
    - Will be discussed in Lecture 8

- **Synchronous Reset**
  - The reset signal is sampled **with respect to the clock**
  - The reset **should be active long enough** to get sampled at the clock edge
  - Results in **completely synchronous circuit**
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk,
                input reset,
                input [3:0] d,
                output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
    if (reset == 0) q <= 0;  // when reset
    else q <= d;            // when clk
end
endmodule
```

- In this example: two events can trigger the process:
  - A *rising edge* on clk
  - A *falling edge* on reset
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk,
               input reset,
               input [3:0] d,
               output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
  if (reset == 0) q <= 0;  // when reset
  else q <= d;            // when clk
end
endmodule
```

- For longer statements, a begin-end pair can be used
  - To improve readability
  - In this example, it was not necessary, but it is a good idea
D Flip-Flop with Asynchronous Reset

```plaintext
module flop_ar (input clk, reset, [3:0] d, output reg [3:0] q);

always @(posedge clk, negedge reset)
begin
  if (reset == 0) q <= 0; // when reset
  else q <= d; // when clk
end
endmodule
```

- First `reset` is checked: if `reset` is 0, `q` is set to 0.
- This is an asynchronous reset as the reset can happen independently of the clock (on the negative edge of reset signal).
- If there is no reset, then regular assignment takes effect.
D Flip-Flop with **Synchronous Reset**

```verilog
module flop_sr (input clk, input reset, input [3:0] d, output reg [3:0] q);

always @(posedge clk)
begin
  if (reset == '0') q <= 0;  // when reset
  else q <= d;            // when clk
end
endmodule
```

- The process is sensitive to only clock
  - Reset *happens only* when the *clock rises*. This is a synchronous reset.
A flip-flop with **enable** and **reset**

- Note that the **en** signal is **not** in the *sensitivity list*
- **q** gets **d** only when **clk** is rising **and** **en** is 1
Example: D Latch

```verilog
module latch (input clk,
              input [3:0] d,
              output reg [3:0] q);

    always @ (clk, d)
    if (clk) q <= d;  // latch is transparent when
     // clock is 1

endmodule
```
Summary: Sequential Statements So Far

- Sequential statements are within an `always` block

- The sequential block is triggered with a change in the sensitivity list

- Signals assigned within an `always` must be declared as `reg`

- We use `<=` for (non-blocking) assignments and do not use `assign` within the always block.
### Basics of `always` Blocks

```verilog
data module example (input clk,
      input [3:0] d,
      output reg [3:0] q);

wire [3:0] normal; // standard wire
reg [3:0] special; // assigned in always

always @(posedge clk)
special <= d; // first FF array

assign normal = ~ special; // simple assignment

always @(posedge clk)
q <= normal; // second FF array
endmodule
```

You can have as many `always` blocks as needed!

Assignment to the same signal in different `always` blocks is not allowed!
Why Does an \textbf{always} Block Remember?

This statement describes what happens to signal $q$

... but what happens when the clock is not rising?

The value of $q$ is preserved (remembered)
An *always* Block Does **NOT** Always Remember

```verilog
module comb (input inv,
            input [3:0] data,
            output reg [3:0] result);

always @ (inv, data) // trigger with inv, data
  if (inv) result <= ~data; // result is inverted data
  else result <= data; // result is data
endmodule
```

- This statement describes what happens to signal `result`
  - When `inv` is 1, `result` is `~data`
  - When `inv` is not 1, `result` is `data`
- The circuit is combinational (no memory)
  - `result` is assigned a value in all cases of the `if .. else` block, always
always Blocks for Combinational Circuits

- An **always** block defines **combinational logic** if:
  - All outputs are always (**continuously**) updated
  1. All right-hand side signals are in the sensitivity list
     - You can use **always @* for short**
  2. All left-hand side signals get assigned in every possible condition of **if .. else** and **case** blocks

- It is easy to make mistakes and **unintentionally describe memorizing elements** (latches)
  - **Vivado** will most likely warn you. Make sure you check the warning messages

- **Always** blocks allow powerful combinational logic statements
  - **if .. else**
  - **case**
Sequential or Combinational?

**Sequential**

```verilog
wire enable, data;
reg out_a, out_b;

always @ (*) begin
    out_a = 1'b0;
    if(enable) begin
        out_a = data;
        out_b = data;
    end
end

No assignment for ~enable
```

**Sequential**

```verilog
wire enable, data;
reg out_a, out_b;

always @ (data) begin
    out_a = 1'b0;
    out_b = 1'b0;
    if(enable) begin
        out_a = data;
        out_b = data;
    end
end

Not in the sensitivity list
```
The **always** Block is **NOT** Always Practical/Nice

```verilog
reg [31:0] result;
wire [31:0] a, b, comb;
wire sel,

always @ (a, b, sel)    // trigger with a, b, sel
  if (sel) result <= a; // result is a
  else result <= b;    // result is b

assign comb = sel ? a : b;
```

- Both statements describe the **same** multiplexer
- In this case, the **always** block is more work
always Block for Case Statements (Handy!)

```verilog
module sevensegment (input [3:0] data, output reg [6:0] segments);

always @ ( * ) // * is short for all signals
    case (data) // case statement
        4'd0: segments = 7'b111_1110; // when data is 0
        4'd1: segments = 7'b011_0000; // when data is 1
        4'd2: segments = 7'b110_1101;
        4'd3: segments = 7'b111_1001;
        4'd4: segments = 7'b011_0011;
        4'd5: segments = 7'b101_1011;
        // etc etc
        default: segments = 7'b000_0000; // required
    endcase

endmodule
```
Summary: *always* Block

- **if .. else** can only be used in *always* blocks

- The *always* block is *combinational* only if all *regs* within the block are always assigned to a signal
  - Use the *default* case to make sure you do not forget an unimplemented case, which may otherwise result in a latch

- Use *casex* statement to be able to check for don’t cares
Non-Blocking and Blocking Assignments

<table>
<thead>
<tr>
<th>Non-blocking ($\leq$)</th>
<th>Blocking ($=$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>always $@ (a)$</td>
<td>always $@ (a)$</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>a $\leq$ 2'b01;</td>
<td>a $= 2'b01;</td>
</tr>
<tr>
<td>b $\leq$ a;</td>
<td>// a is 2'b01</td>
</tr>
<tr>
<td>// all assignments are made here</td>
<td>b = a;</td>
</tr>
<tr>
<td>// b is not (yet) 2'b01</td>
<td>// b is now 2'b01 as well</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

- All assignments are made at the end of the block
- All assignments are made in parallel, process flow is not-blocked
- Each assignment is made immediately
- Process waits until the first assignment is complete, it blocks progress
Example: Blocking Assignment

- Assume all inputs are initially ‘0’

```verbatim
always @ ( * )
begin
    p    = a ^ b ;    // p    = 0 1
    g    = a & b ;    // g    = 0 0
    s    = p ^ cin ;  // s    = 0 1
    cout = g | (p & cin); // cout = 0 0
end
```

- If a changes to ‘1’
  - All values are updated in order
The Same Example: Non-Blocking Assignment

- Assume all inputs are initially ‘0’

```verilog
always @( * )
    begin
        p    <= a ^ b ;    // p    = 0 1
        g    <= a & b ;    // g    = 0 0
        s    <= p ^ cin ;  // s    = 0 0
        cout <= g | (p & cin) ;  // cout = 0 0
    end
```

- If a changes to ‘1’
  - All assignments are concurrent
  - When s is being assigned, p is still 0
The Same Example: Non-Blocking Assignment

- After the first iteration, \( p \) has changed to ‘1’ as well

```verbatim
always @( *)
begin
    p    <= a ^ b ;   // p = 1 1
    g    <= a & b ;   // g = 0 0
    s    <= p ^ cin ; // s = 0 1
    cout <= g | (p & cin) ; // cout = 0 0
end
```

- Since there is a change in \( p \), the process triggers again
- This time \( s \) is calculated with \( p=1 \)
Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic.

```verilog
always @ (posedge clk)
q <= d; // non-blocking
```

- Use continuous assignments (`assign`) to model simple combinational logic.

```verilog
assign y = a & b;
```
Rules for Signal Assignment (Cont.)

- Use `always @ (*)` and `blocking` assignments (`=`) to model more complicated combinational logic.

- You **cannot** make assignments to the same signal in more than one `always` block or in a `continuous assignment`

```
always @ (*)
a = b;

always @ (*)
a = c;
assign a = c;
```
Recall: Finite State Machines (FSMs)

- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic
Recall: Finite State Machines (FSMs) Comprise

- **Sequential circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge

- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be
  - Output logic
    - Generates the outputs
FSM Example 1: Divide the Clock Frequency by 3

The output $Y$ is HIGH for one clock cycle out of every 3. In other words, the output divides the frequency of the clock by 3.
Implementing FSM Example 1: Definitions

```verilog
module divideby3FSM (input clk,
                     input reset,
                     output q);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
```

- We define **state** and **nextstate** as 2-bit reg
- The parameter descriptions are **optional**, it makes reading easier
Implementing FSM Example 1: State Register

```vhdl
// state register
always @(posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
```

- This part defines the state register (memorizing process)
- Sensitive to only `clk`, `reset`
- In this example, `reset` is active when it is `1` (active-high)
Implementing FSM Example 1: Next State Logic

// next state logic
always @ (*)
case (state)
  S0:    nextstate = S1;
  S1:    nextstate = S2;
  S2:    nextstate = S0;
  default: nextstate = S0;
endcase
Implementing FSM Example 1: Output Logic

// output logic
assign q = (state == S0);

- In this example, output depends only on state
  - **Moore type FSM**
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

    always @ (posedge clk, posedge reset) // state register
        if (reset) state <= S0;
        else state <= nextstate;

    always @ (*) // next state logic
        case (state)
            S0:    nextstate = S1;
            S1:    nextstate = S2;
            S2:    nextstate = S0;
            default: nextstate = S0;
        endcase

    assign q = (state == S0); // output logic
endmodule
FSM Example 2: Smiling Snail

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1’s and 0’s on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail’s brain.
module SmilingSnail (input clk,
    input reset,
    input number,
    output smile);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
parameter S3 = 2'b11;
Implementing FSM Example 2: State Register

```
// state register
always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;
```

- This part defines the **state register** (memorizing process)
- Sensitive to only **clk, reset**
- In this example **reset** is active when ‘1’ (active-high)
Implementing FSM Example 2: Next State Logic

// next state logic
always @ (*)
    case (state)
        S0: if (number) nextstate = S1;
            else nextstate = S0;
        S1: if (number) nextstate = S2;
            else nextstate = S0;
        S2: if (number) nextstate = S2;
            else nextstate = S3;
        S3: if (number) nextstate = S1;
            else nextstate = S0;
        default: nextstate = S0;
    endcase
Implementing FSM Example 2: Output Logic

```plaintext
// output logic
assign smile = (number & state == S3);
```

- In this example, output depends on state and input
  - **Mealy type FSM**
- We used a simple combinational assignment
module SmilingSnail (input clk, 
    input reset, 
    input number, 
    output smile);

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    // state register
    always @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;

    always @ (*) // next state logic
        case (state)
            S0: if (number)
                nextstate = S1;
            else nextstate = S0;
            S1: if (number)
                nextstate = S2;
            else nextstate = S0;
            S2: if (number)
                nextstate = S2;
            else nextstate = S3;
            S3: if (number)
                nextstate = S1;
            else nextstate = S0;
            default: nextstate = S0;
        endcase

    assign smile = (number & state==S3);

endmodule
What Did We Learn?

- Basics of defining **sequential circuits** in Verilog

- The **always** statement
  - Needed for defining memorizing elements (**flip-flops, latches**)
  - Can also be used to define **combinational circuits**

- **Blocking** vs **Non-blocking** statements
  - `=` assigns the value **immediately**
  - `<=` assigns the value **at the end of the block**

- **Writing FSMs**
  - Next state logic
  - State assignment
  - Output logic
Next Lecture:
Timing and Verification
Lecture 7: Hardware Description Languages and Verilog

Prof. Onur Mutlu

ETH Zürich
Spring 2021
18 March 2021