

Logic/Verilog Workshop

Make sure to sign in!

Breakdown of Workshop

PART 1: Logic

Logic Essentials - Introduction, Boolean logic/operations, Types of Logic

PART 2: Verilog

Verilog Basics - Intro, Basics blocks/control flow, programming practice!

Checkpoint: Any expected prerequisites?

- *Some programming experience (have taken/taking CS 31)*
- *Human logic*
- *Digital design logic is a plus, but we'll teach what you need for now :D*
- *Excitement for a super cool part of the EE/CS world*

What is Logic (Quick intro/review)?

- Foundation of any computer system
 - Any operation can be broken down into block that represent simply yes or no questions
 - In the engineering world \rightarrow No = 0, Yes = 1
- Needed to design circuits -two subcategories
 - Combinational Logic
 - Sequential Logic
- Must understand logic to design/program hardware
 - Hardware description language
 - Hardware - processors, FPGA boards

A Bit of Binary

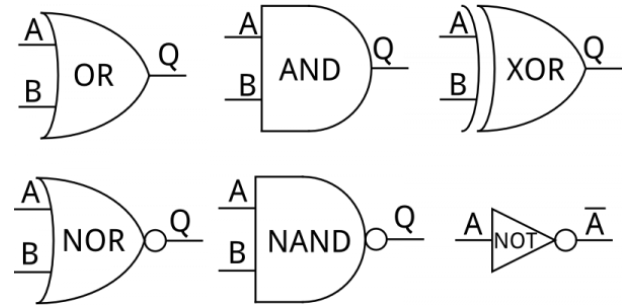
- Binary is Base-2 representation
 - Any digit in representation is 0 or 1
 - **Binary digit = BIT**
- A machine or a part of the machine can have the state ON or OFF
 - We like number so ON is 1 and OFF is 0
 - Analogous to YES and NO
- Logic Programming can work on single bits or a string of bits

Length	Name	Example
1	Bit	0
4	Nibble	1011
8	Byte	10110101

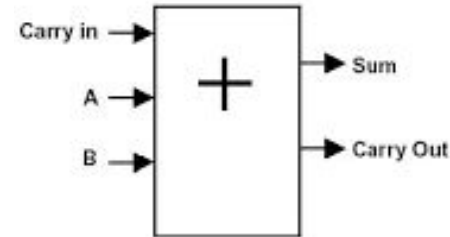
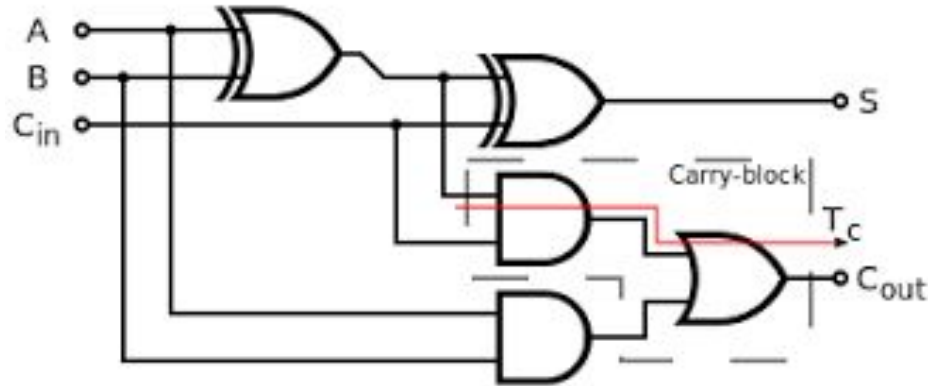
1 0 0 1 1 0 1 1
| | | | | | | |
 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0

Combinational Logic

- Boolean Logic
 - AND gate - output is 1 if BOTH inputs are 1
 - OR gate - output is 1 if AT LEAST one input is 1
 - XOR gate - output is 1 if ONLY one input is 1
 - NAND gate - output is 1 if AT LEAST one input is 0
 - NOR gate - output is 1 if BOTH inputs are 0
 - NOT gate - inverts input ($0 \rightarrow 1$; $1 \rightarrow 0$)
- Combine boolean logic to create other operations (e.g arithmetic)
 - Adders
 - Bottom line: the art of **combining** blocks/gates to produce an output



Combinational Logic (Adder)



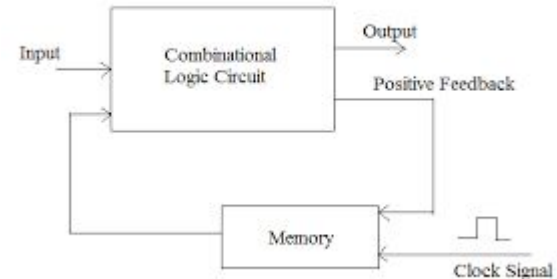
Carry-in	A	B	Carry-Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sequential Logic

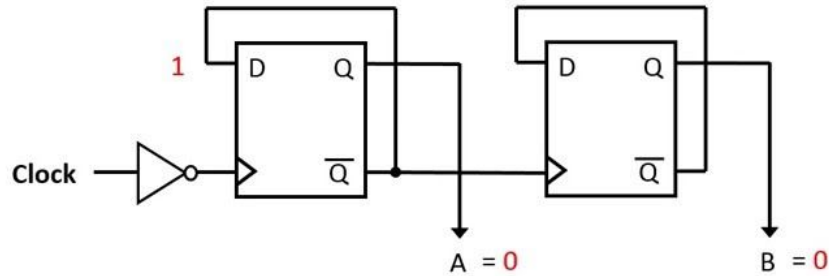
- Computers made to learn and execute
 - Needs memory/data to build on
 - An operation takes more than one run for the correct results
 - CS analogy (CAUTION)
- Using the same block again via a **feedback loop**
- How do we store memory in a circuit
 - Data that is held to modify goes into a **hardware register (D-flip flop)**
 - Ensures we caught the data the first time and it's there for us to use next time
- Bottom line: Sequentially modify data through blocks to produce an output

```
count = 0;  
count = 0+1; //1  
count = 1 + 1; // 2  
count = 2 + 1; //3  
count = 3 + 1; //4
```

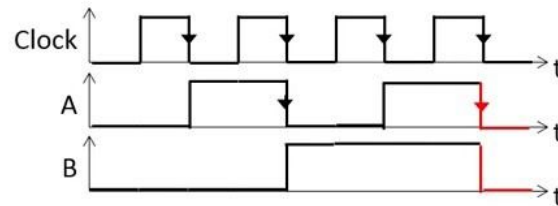
```
count = 0;  
for (int i = 0; i < 4; i++)  
    count += 1;
```



Sequential Logic (Counter)



Clock pulse number	B	A
0	0	0
1	0	1
2	1	0
3	1	1
4	0	0



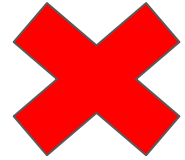
What is a Hardware Description Language?

- Implementing Logic is multidisciplinary
- Hardware engineers use this to model/simulate digital electronics
 - Verify logic of a designed circuit
 - Example: ASICs, FPGAs
- Popular languages: Verilog and VHDL
- We will be using Verilog

Verilog

- Like C but NOT C
 - Case sensitive
 - Same style comments
 - Operators (+,-, &,~,etc)
- First thing to remember and never forget
 - Code is executes ALL AT ONCE
 - NOT LINE-BY-LINE (please repeat)

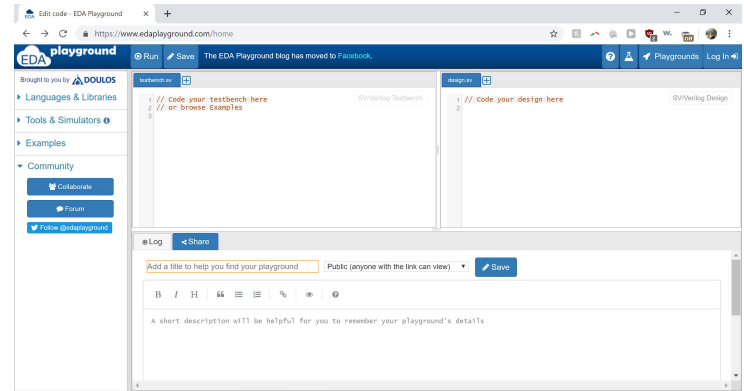
```
count = 0;  
count = 0+1; //1
```



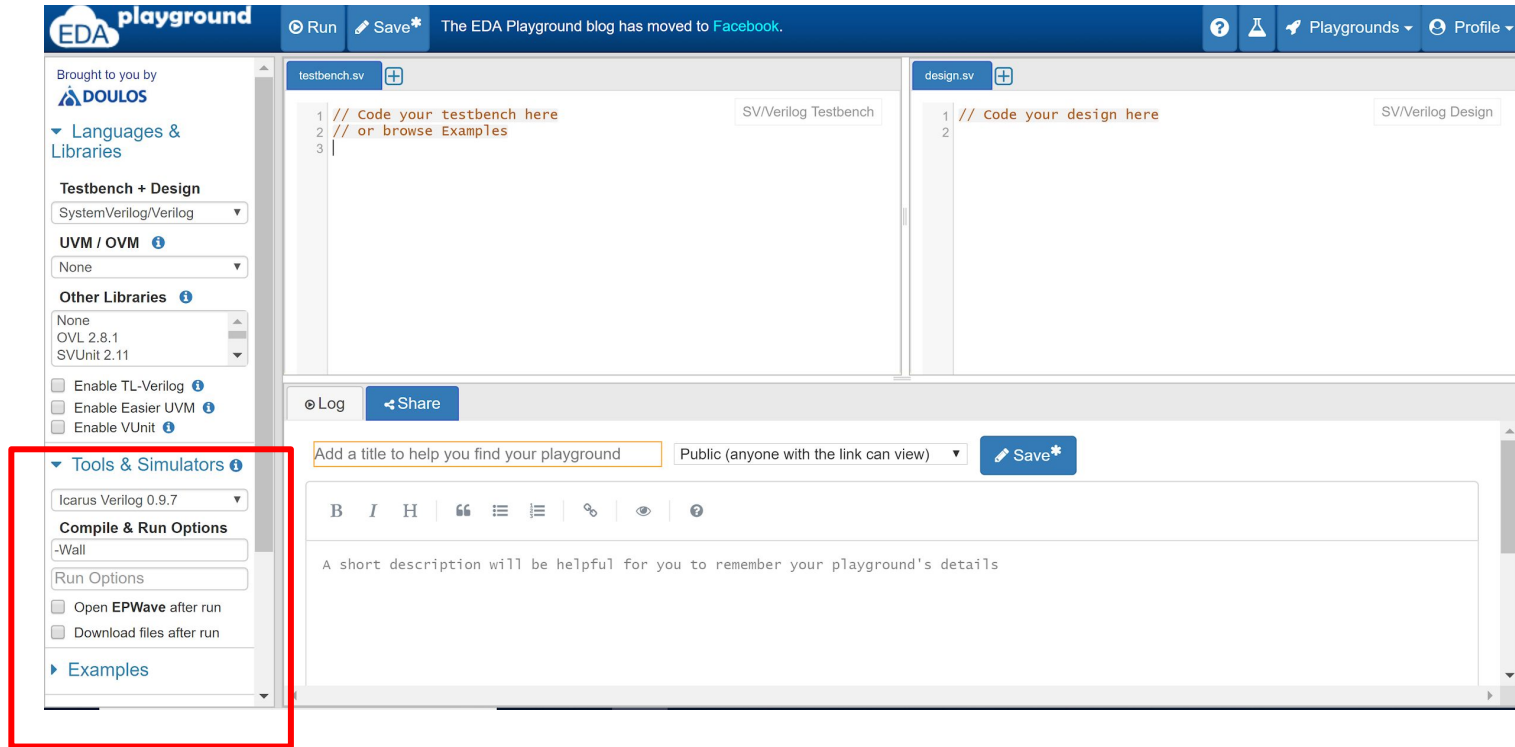
```
reg [31:0] cnt;  
initial begin  
    cnt <= 32'b0  
end  
  
always @(posedge clk)  
begin  
    cnt <= cnt + 1  
end
```

EDA Playground

- Online environment for Verilog
- Login
 - Facebook/Google/Register
- Configure environment
 - Tools & Simulators → Icarus Verilog 0.9.7
 - Libraries
 - SystemVerilog/Verilog
 - Everything else is None/Unselected
- Right side: Verilog File Editor
- Left Side: Testbench
 - Syntax you don't have to worry about today



EDA Playground Settings



The screenshot displays the EDA Playground web interface. On the left sidebar, the 'Tools & Simulators' section is highlighted with a red box. This section includes a dropdown for 'Icarus Verilog 0.9.7', a 'Compile & Run Options' section with a '-Wall' option, and checkboxes for 'Open EPWave after run' and 'Download files after run'. Below these are links for 'Examples' and 'Languages & Libraries'. The main area features two code editors: 'testbench.sv' and 'design.sv'. The 'testbench.sv' editor contains the following code:

```
1 // Code your testbench here
2 // or browse Examples
3
```

The 'design.sv' editor contains the following code:

```
1 // Code your design here
2
```

Below the code editors is a 'Log' section and a 'Share' button. The 'Share' section includes a title input field with the placeholder 'Add a title to help you find your playground', a visibility dropdown set to 'Public (anyone with the link can view)', and a 'Save*' button. Below the share section is a rich text editor with formatting options (B, I, H, quote, list, link, image, help) and a description area with the placeholder text 'A short description will be helpful for you to remember your playground's details'.

Numbers

- Conventional written as
 - `[num_bits]'[base-system][value]`
- Examples
 - `5'd13`
 - 5 bits, decimal representation
 - `5'b1101` or `5'b13`
 - 5 bits, binary representation
- Unsigned unless explicitly made to be signed (`2'sb11` is -1)

Verilog Data types : Register

- NOT the physical registers/D-flip flop that we saw earlier
- variables that hold data that Verilog named a “register”
- Use for non-continuous assignment
- Both types of logic use registers
- Example: `reg = 2'b0`

Verilog Data type: Wires

- Type of *net*: connect between elements
- Holds **transient** values (continuously driven)
- Assigned values continuously
- Basically think of a physical wire (just there to hold the current but you don't care about how much until it gets to a component)
- input/output ports of a module \Rightarrow wires!
- Wires vs Reg Resource:

<https://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf>

Operators

- Bitwise
 - NOT (~), AND(&), OR(|), XOR(^), XNOR(~^)
 - Reduction: To use on a single operand
 - $\sim 4'b1001 = 1'b0$
- Logical
 - !, &&, ||
 - One bit result
- Arithmetic
 - +, -, *, /, %
 - Slower → Think of them as built-in implemented module that are called

Operators

- Shift
 - `<<` , `>>` (logical)
 - `<<<` , `>>>` (arithmetic)
- Concatenation
 - `{a,b}`
 - Helpful when you want to work on parts of an input in parallel
- Replication
 - `{n{m}}` → helpful for extending 0 bits or 1111...111 values
- Assignment
 - `=` (blocking) : immediate and happens first, pure CL
 - `<=` (non-blocking) : Waits until all RHS have been evaluated; SL or CL/SL

Structural Verilog

- Creating the basic components using raw operators/structures
- Logic gates, adders, muxes, etc

```
module xor_gate(a,b,y);  
    input a, b;  
    output y;  
    assign #1 y = a ^ b;  
endmodule
```

```
module and_gate(a,b,y);  
    input a, b;  
    output y;  
    assign #1 y = a & b;  
endmodule
```

```
module structural_adder(A, B, sum,  
    carry);  
    input A, B;  
    output sum, carry;  
    xor_gate xor_gate1(A,B,sum);  
    and_gate and_gate1(A,B,carry);  
endmodule
```

Behavioral Verilog

- Ultimate functionality → algorithm-like
- Continuous

```
wire sum, carry;  
assign sum = A ^ B;  
assign carry = A & B;
```

- Procedural

```
reg sum, carry;  
always @ (A)  
begin  
    sum <= A + B;  
    carry <= A & B;  
end
```

Modules

- Building blocks of Verilog
 - Name is what the purpose it → to make functionality modular
- Must declare which are inputs, which are outputs

```
module top(a, b, ci, s, co);  
    input a, b, ci;  
    output s, co;  
    wire s;  
    reg g, p, co;  
    assign s = a ^ b ^ ci;  
    // combinatorial always  
    // block using begin/end  
    always @* begin  
        g = a & b;  
        p = a | b;  
        co = g | (p & ci);  
    end  
endmodule
```

Always Block

- Sensitivity list (@ __)
 - Always do this block of code when this happens
 - always @ (posedge clk or posedge reset)
 - Basically when there's a change to what's in ()
- Needed to wrapping around and looping structure

```
always @* begin
    g = a & b;
    p = a | b;
    co = g | (p & ci);
end
```

Case Statements

- Kind of like switch statements
- Put in always blocks
- Useful to implement multiplexers w/ more than one output
 - Maps inputs to specific outputs

```
always @(*)
  z = 4'bx; (#1)
  case (f)
    0: z = a + b;
    1: z = a * b;
    2: z = a & b;
    3: z = a | b;
    default: z = 4'bx; (#2)
  endcase
```

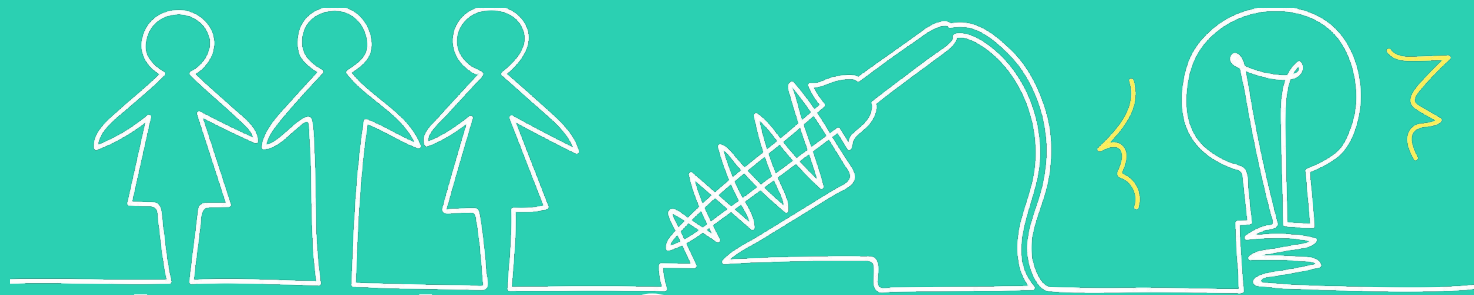
Summary

- New type of programming for most
- Know/design your logic before coding
- Debugging is a unique process for Verilog
- It's worth it in the future
 - World of FPGAs, ASICs, etc
 - EE/Defense companies love modelling hardware

Resources/Next Steps

- Logic Crash Course: <https://learn.sparkfun.com/tutorials/digital-logic/all>
- Wires vs. Reg: <https://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf>
- Extremely verbose tutorial:
http://www.referencedesigner.com/tutorials/verilog/verilog_01.php
- On campus
 - ECE M16/CS M51A (Logic class -required)
 - ECE M 116L/CS M152A (Digital Design Lab - Program FPGAs!)**
 - CS M152B (CSE “capstone”/elective)
 - HKN Workshops**

**Resources/Materials adopted as guidelines for flow!



Thanks for coming!

(And congrats on completing Week 4 omg)