Verilog
For Computer Design

CS/ECE 552, Fall 2020

Guanzhou Hu

Based on slides from

Prof. Karu Sankaralingam (UW-Madison),

Derek Hower (UW-Madison), Andy Phelphs (UW-Madison) and Prof. Milo Martin (University of Pennsylvania)

Overview

- Why Verilog?

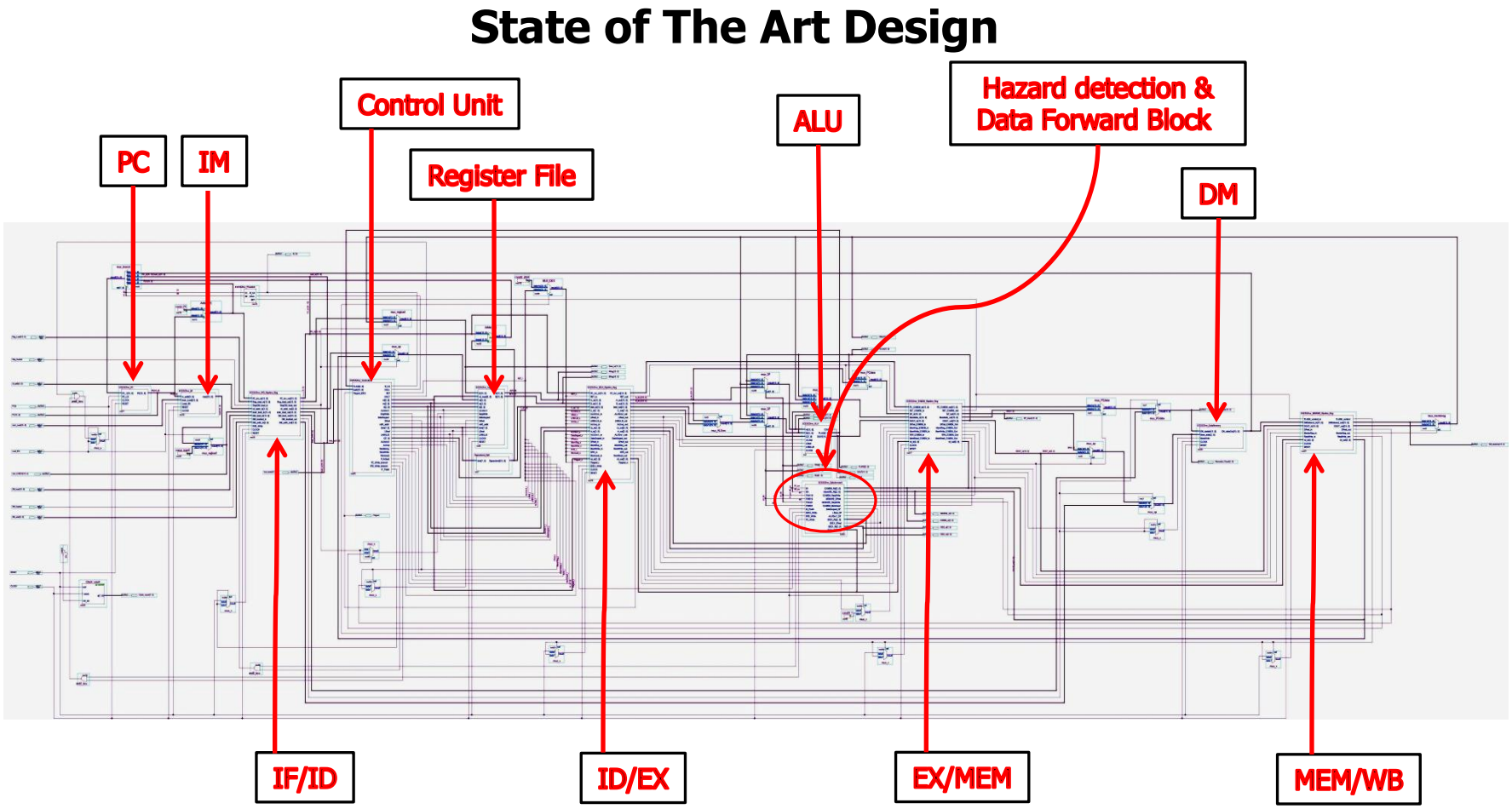
  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Demo Walkthrough

Overview

- Why Verilog?

  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Demo Walkthrough

Why Verilog and Why Not Manual Design?



Do you want to design this Processor manually?

Hardware Description Languages (HDLs)

- Textual representation of a digital logic design
- HDLs are NOT "programming languages"
  - A procedural programming lang defines a sequence of events for the processor to execute one-by-one
  - An HDL describes what a chip looks like: what are the components and how they are wired together
  - For many people, a difficult conceptual leap
- Similar development chain
  - Compiler: source code → assembly code → binary machine code
  - Synthesis tool: HDL source → gate-level specification → hardware

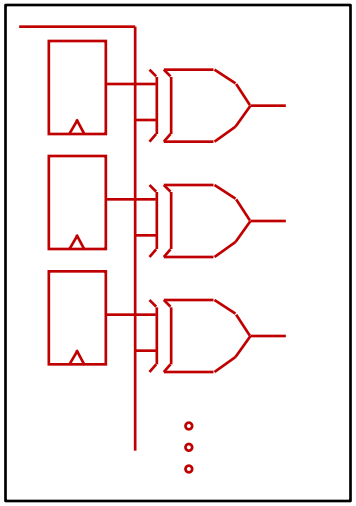
Why an HDL is not a Programming Language

- In a software program, we start at the beginning (e.g. "main"), and we proceed sequentially through the code as directed
- The program represents an algorithm, a step-by-step sequence of actions to solve some problem

```
for (i = 0; i < 10; i++) {
    if (newPattern == oldPattern[i])
        match[i] = true;
}
```

Why an HDL is not a Programming Language

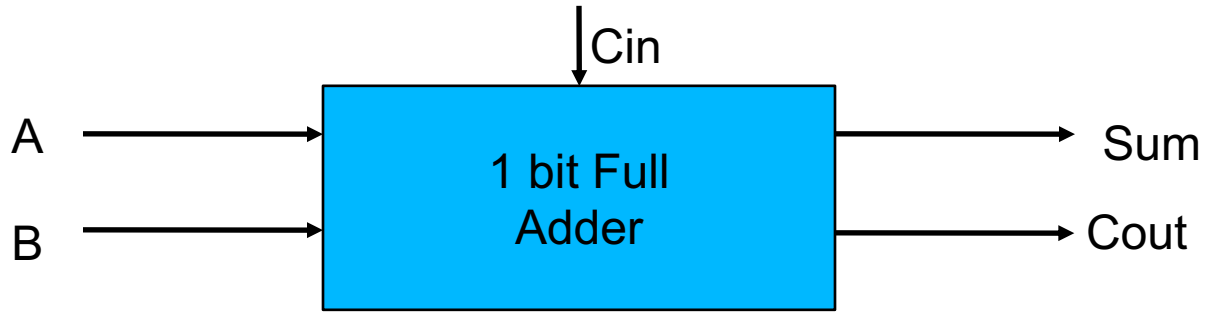
- Hardware is all active at once; there is no starting point
- It is a static layout of logic circuits



Starting With an Example…

```
module fulladd (input A, B, Cin,
                output sum, Cout );

    assign sum = A ^ B ^ Cin;
    assign Cout = (A & B)| (A & Cin)| (B & Cin);

endmodule
```

Synthesis

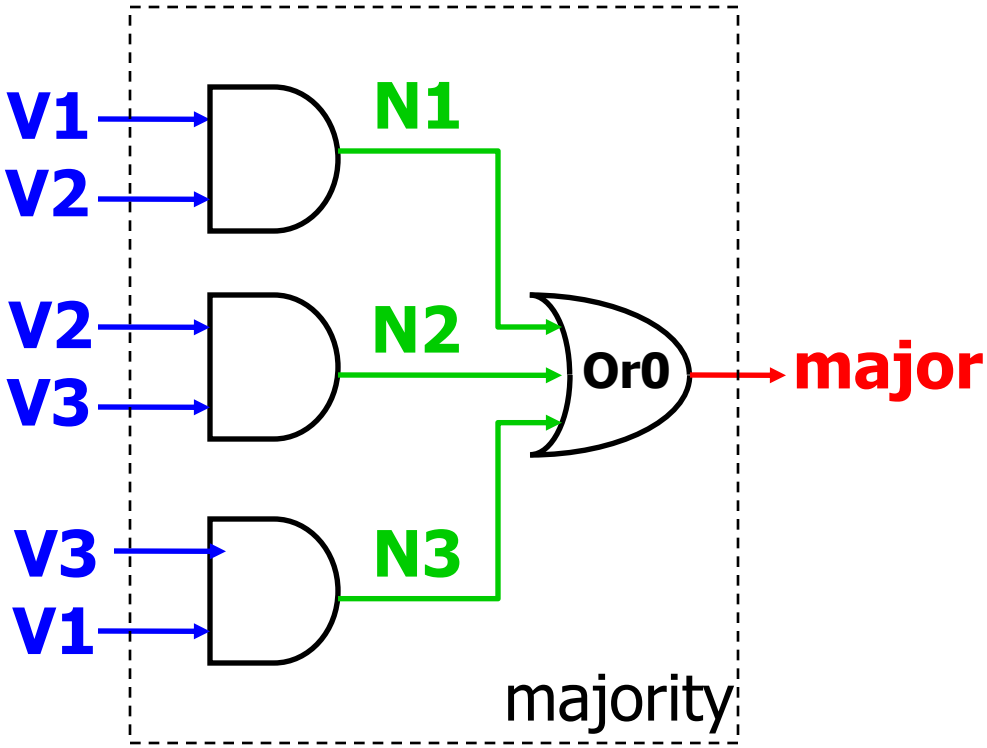


HDL Coding Constructs

- *Structural* constructs specify actual hardware structures
  - Low-level, direct correspondence to hardware
    - Primitive gates (e.g., and, or, not)
    - Hierarchical structures via modules
- *RTL/Dataflow* constructs specify an operation on bits
  - High-level, more abstract
    - Specified via equations, e.g., out = (a & b) | c
- *Behavioral* – Describes behavior of the circuit
  - Always, initial blocks, procedural assignments
  - Not all behavioral constructs are synthesizable
    - Even some combinational logic won't synthesize well
    - out = a % b // modulo op – what does this synthesize to?

Structural Example

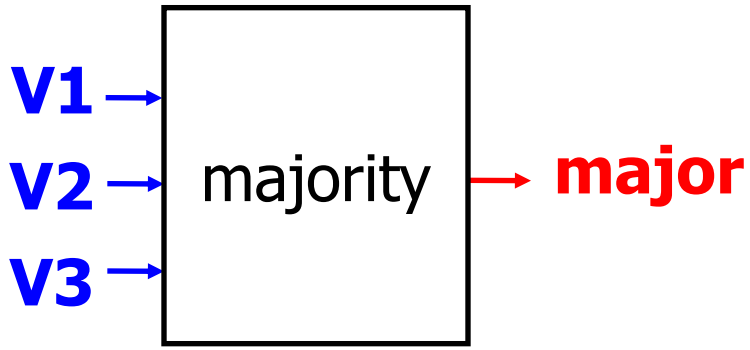
```
module majority (major, V1, V2, V3) ;

    output major ;
    input V1, V2, V3 ;

    wire N1, N2, N3;

    and A0 (N1, V1, V2),
        A1 (N2, V2, V3),
        A2 (N3, V3, V1);

    or  Or0   (major, N1, N2, N3);

endmodule
```



RTL/Dataflow Example

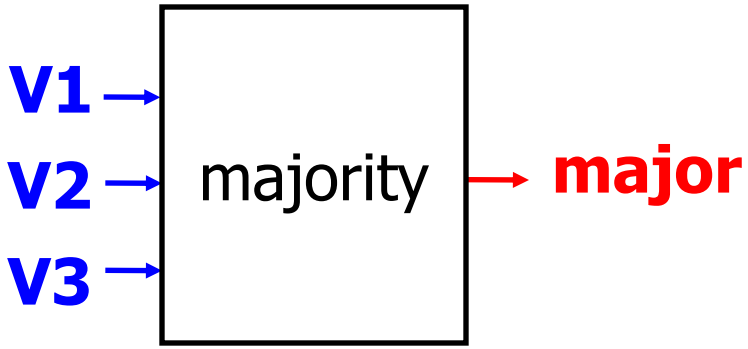
Continuous Assignment Statement

```
module majority (major, V1, V2, V3) ;

    output major ;
    input V1, V2, V3 ;

    assign major = V1 & V2
                 | V2 & V3
                 | V1 & V3;
endmodule
```



Behavioral Example

```
module majority (major, V1, V2, V3) ;

    output reg major ;
    input V1, V2, V3 ;

    always @(V1, V2, V3) begin
        if (V1 && V2 || V2 && V3
          || V1 && V3) major = 1;
        else major = 0;
    end

endmodule
```



Overview

- Why Verilog?

  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Demo Walkthrough

Recall: Two Types of Digital Circuits

- Combinational Logic
  - Logic without state variables
  - Examples: adders, multiplexers, decoders, encoders
  - No clock involved
  - Not edge-triggered
  - All "inputs" are triggers
- Sequential Logic (details explained later)
  - Logic with state variables
  - State variables: registers (latches, flip-flops), memory
  - Clocked - Edge-triggered by clock signal
  - State machines, multi-cycle arithmetic, processors
  - Only clock (and possibly reset) appear in trigger list
  - Can include combinational logic that feeds the register

Number Representation

Format: <size><base_format><number>

Examples:

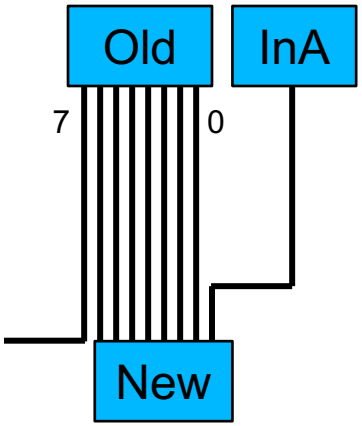
6'b010_111	gives 010111
8'b0110	gives 00000110
8'b1110	gives 00001110
4'bx01	gives xx01
16'H3AB	gives 0000001110101011
24	gives 0…0011000
5'O36	gives 11100
16'Hx	gives xxxxxxxxxxxxxxxx
8'hz	gives zzzzzzzz

Compose Wider Signal using Brackets

Examples:

{4'hA, 4{1'b1}} gives 8'b10101111

{Old[6:0], InA} gives a 8-bit wire New like:



Module Definition

```
module not1 (in1, out);
   input  in1;
   output out;

   assign out = ~in1;
endmodule
```

- In all HWs and projects, only allowed to use a very basic set of Verilog (see Verilog rules of this course)
- In HW1, we will provide basic modules such as the NOT gate above; Instantiate them to construct your modules

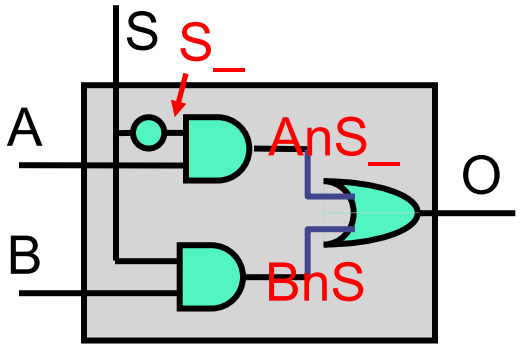
Module Instantiation: Hierarchical Design

```
module not1_2 (In, Out);
   input  [1:0] In;
   output [1:0] Out;

   not1 n0 (.in1(In[0]), .out(Out[0]));
   not1 n1 (.in1(In[1]), .out(Out[1]));
endmodule
```

- Build up more complex modules using simpler modules
  - The idea of **Abstraction**!
- Rule: MUST use explicit port name mapping
- Example: 2-bit wide NOT gate from two 1-bit gates

Verilog "wire"

```
module mux2to1 (
    input  S, A, B,
    output Out );

    wire S_, AnS_, BnS;

    not (.in1(S), .out(S_));
    and (.in1(S_), .in2(A), .out(AnS_));
    and (.in1(S), .in2(B), .out(BnS));
    or  (.in1(AnS_), .in2(BnS), .out(Out));
endmodule
```



- Give names to internal wires in your layout

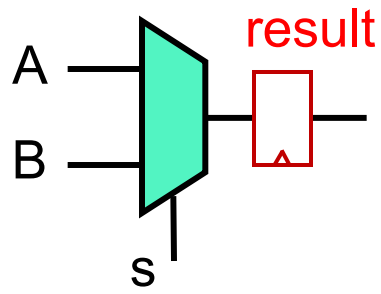
Wire Assignment

- Wire assignment: "continuous assignment"
  - **Order of statements not important to Verilog**, executed totally in parallel, describes the same hardware
  - But order of statements can be important to clarity of thought!
  - When right-hand-side changes, it immediately flows through to left
  - Designated by the keyword **`assign`**

```
wire [3:0] c;
assign c = a | b;
wire [3:0] c = a | b;      // same thing
```

Verilog "reg"

```
reg result;
always @ (s or A or B) begin
   case(s)
      1'b1: result = A;
      1'b0: result = B;
      default: result = 1'bx;
   endcase
end
```



- Think of a reg variable as a register on a wire

When to Use `wire` and When `reg`!

- Wire
  - ✓ Module declaration: Inputs(Yes), Outputs (Yes)
  - ✓ Module instantiation: Connect input and output ports
  - ✓ Must be driven by something, cannot store values
  - ✓ Only legal type on left side of an assign statement
  - ✓ Not allowed on left side of = or <= in an always@ block
  - ✓ Most of the times combinational logic
- Reg
  - ✓ Module instantiation: Input port (Yes) , Output Port (No)
  - ✓ Module declaration: Inputs(No), Outputs (Yes)
  - ✓ Only legal type on left side of = or <= in an always@ block
  - ✓ Only legal type on left side of initial block (test bench)
  - ✓ Not Allowed on left side of an assign statement
  - ✓ Used for both sequential and combinational logic

Operators

- On wires:
  - & (and), | (or), ~ (not), ^ (xor)
- On vectors:
  - &, |, ~, ^ (bit-wise operation on all wires in vector)
    - E.g., `assign vec1 = vec2 & vec3;`
  - &, |, ^ (reduction on the vector)
    - E.g., `assign wire1 = | vec1;`
  - ==, != (equality); ===, !== (identity)
  - M << const, M >> const (shift by const bits)
- Can be arbitrarily nested

Conditional Operator

- Verilog supports the ? : ternary operator

Examples:

```
assign out = S ? B : A;

assign out = sel == 2'b00 ? a :
             sel == 2'b01 ? b :
             sel == 2'b10 ? c :
             sel == 2'b11 ? d : 1'b0;
```

What do these do?

Parameters

- Parameters

```
module mux2to1_N(Sel, A, B, O);
    parameter N = 1
    input [N-1:0] A;
    …

mux2to1_N #(4) mux1 (…
```

Verilog Pre-processor

- Using macros
  - Constants: `` `define ``

```
`define letter_A 8'h41
wire w = `letter_A;
```

  - File inclusion: `` `include ``

- Rule: define all constants in `module_name_config.v` and include this file in your module

Non-binary Hardware Values

- A hardware signal can have four values
  - `0`, `1`
  - `x`: don't know, don't care
  - `z`: high-impedance (no current flowing)
- Two meanings of "x"
  - Simulator indicating an unknown state
  - Or: You telling synthesis tool you don't care
    - Synthesis tool makes the most convenient circuit (fast, small)
    - Use with care, leads to synthesis dependent operation
- Uses for "z"
  - Tri-state devices drive a zero, one, or nothing (z)
  - Many tri-states drive the same wire, all but one must be "z"
    - Example: multiplexer

Case Statements

```
case (<expr>)
    <match-constant1>: <stmt>
    <match-constant2>: begin
          <stmt>
      end
    <match-constant3>,<match-constant4>: <stmt>
    default: <stmt>
endcase
```

- Also have `casez` / `casex` for wildcards

Case Statements

- Useful to make big muxes
  - Very useful for "next-state" logic
  - BUT they are easy to abuse
- If you don't set a value, it retains its previous state
  - Which is a latch!
- We will allow case statements, but with some severe restrictions:
  - Every value is set in every case
  - Every possible combination of select inputs must be covered
  - MUST have default case
  - Each case lives in its own "`always`" block, sensitive to changes in all of its input signals
  - This is our ONLY use of "`always`" and "`reg`"

System Tasks

- Start with `$`
  - For output:
    - `$display`
    - `$fdisplay`
    - `$monitor`
    - `$dumpvars`
  - Internal Clock: `$time`
  - Finish simulation: `$finish`
  - Pause for debugging: `$stop`
  - Direct manipulation of memory:
    - `$readmemh`
    - `$writememh`

Everything about Verilog for this Course

1. Only allowed to use a very basic set of Verilog; see Verilog rules
2. Verilog cheatsheet by Karu as a quick reference of syntax; also includes the rules in it
3. Additional filename convention rules: Exactly one module per file, file named `module_name.v`

Ask TA or Professor if you are experiencing any difficulty in following these guidelines. We are glad to help!

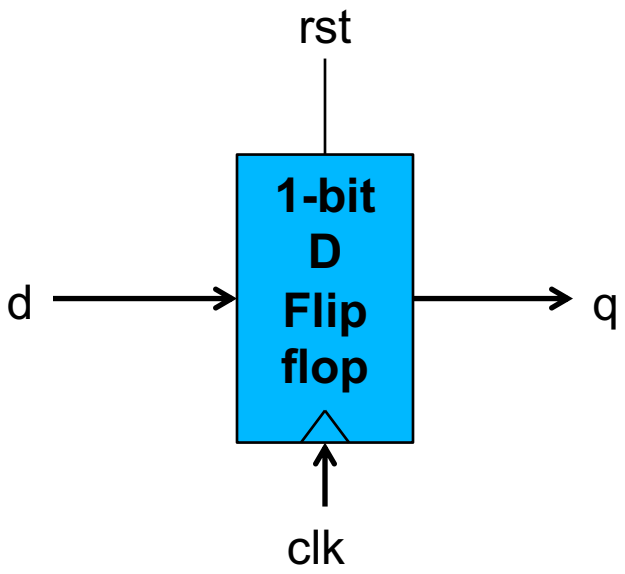
Overview

- Why Verilog?

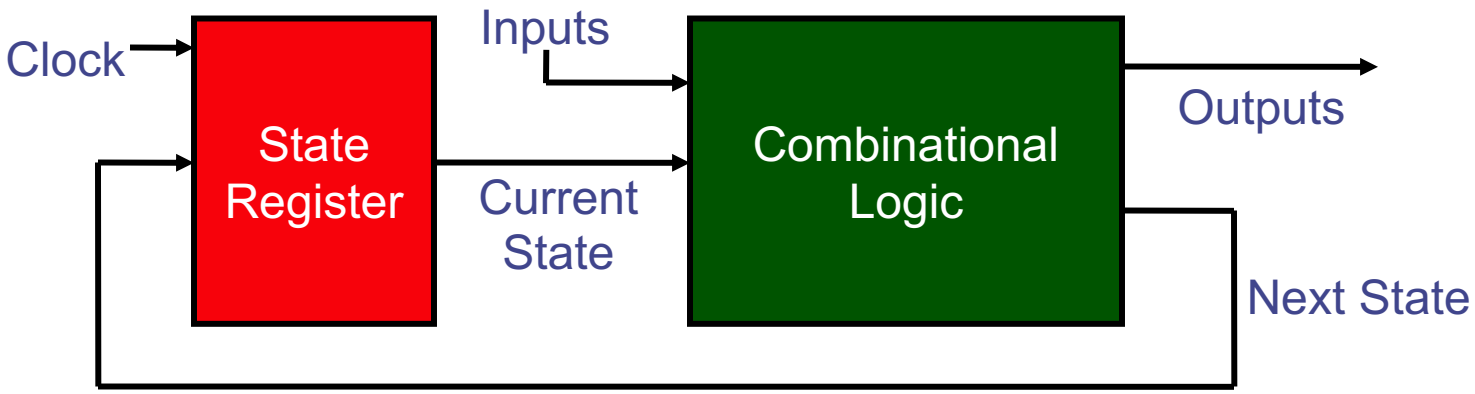
  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Case Study, Verilog Tools and Demo

Sequential Logic in Verilog

- Use the `dff` module (1-bit FF) provided to create wider FFs, then use them as state registers
  - NO direct use of Verilog "`reg`"



Example: State Machine



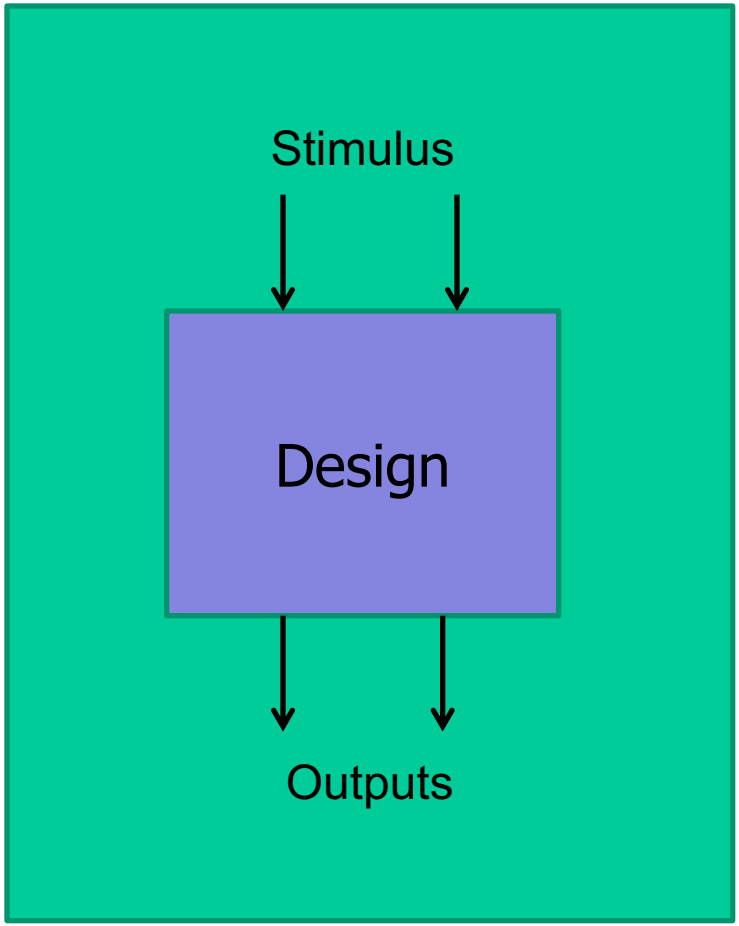
- State Register is your n-bit FF built from `dff`
- Separating combinational logic from sequential state elements is a good design practice

Overview

- Why Verilog?

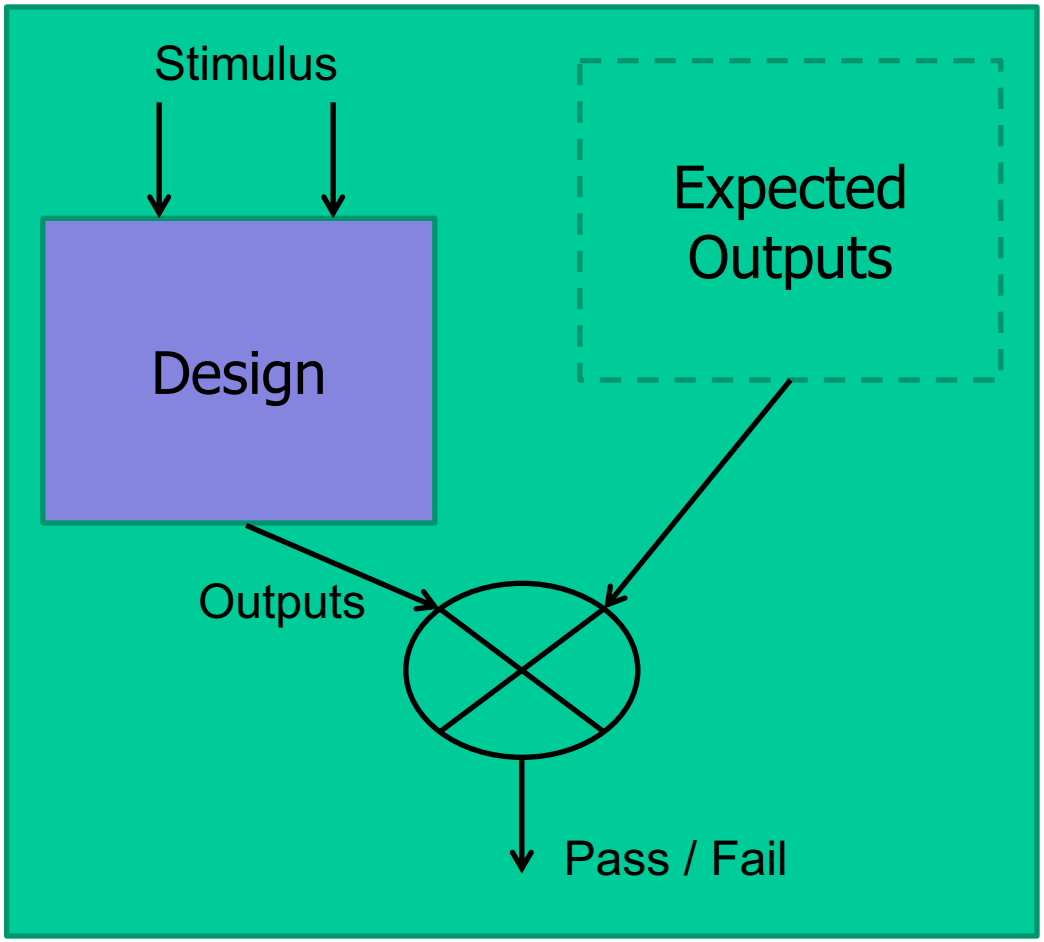
  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Demo Walkthrough

Testbench – For Simple Homework

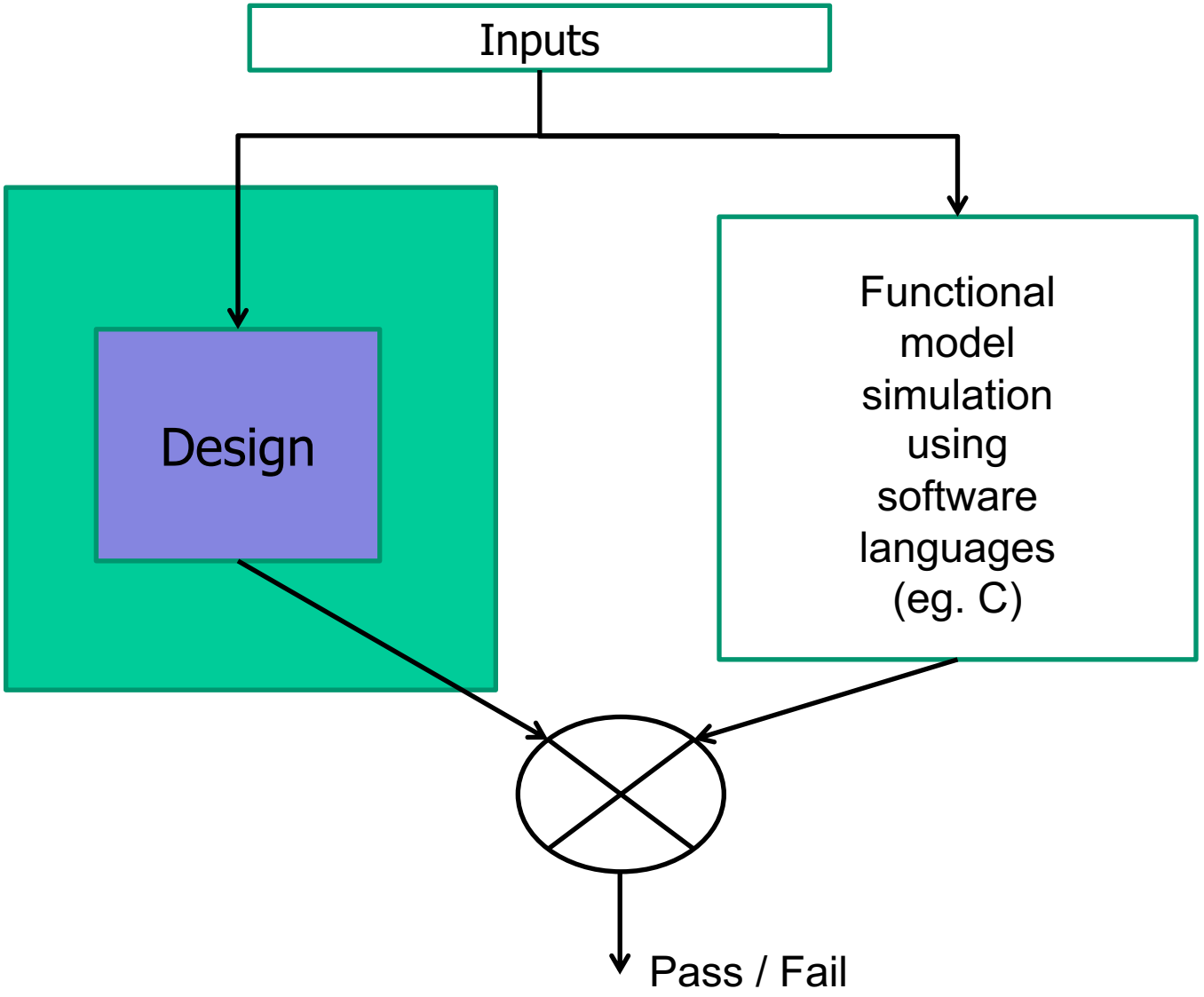


And “visually” inspect the outputs…

Testbench – w/ Expected Outputs



Testbench – For Course Projects



Overview

- Why Verilog?

  High-level Description of Verilog
- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and Instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
- Sequential Logic
- Testbench Structure
- Demo Walkthrough

Demo Walkthrough of HW Problem

Check the pinned Piazza note:

https://canvas.wisc.edu/courses/205192/external_tools/65

- I will show you a pure command-line walkthrough now
- For graphical ModelSim dev/debugging, you may connect to a CSL machine or use a local installation
  - Just be sure to put the finished work onto a CSL machine and run a final check before submission