

# Design of Digital Circuits and Systems

## SystemVerilog Review & Tips

**Instructor:** Vikram Iyer

**Teaching Assistants:**

Ariel Kao

Josh Wentzien

Selim Saridede

Jared Yoder

Derek Thorp

# Lecture Outline

- ❖ **Course Introduction**
- ❖ Course Policies
- ❖ Hardware Description Language
- ❖ SystemVerilog Review & Tips

# Introductions: Course Staff

- ❖ Your Instructor: just call me Vikram
  - CSE Assistant Professor
  - Houston TX → Berkeley Undergrad → UW PhD
- ❖ TAs: Ariel Kao, Josh Wentzien, Selim Saridede, Jared Yoder, Derek Thorp
  - Available in lecture, office hours, and discussion board
  - An invaluable source of information and help
- ❖ Get to know us – we are here to help you succeed!

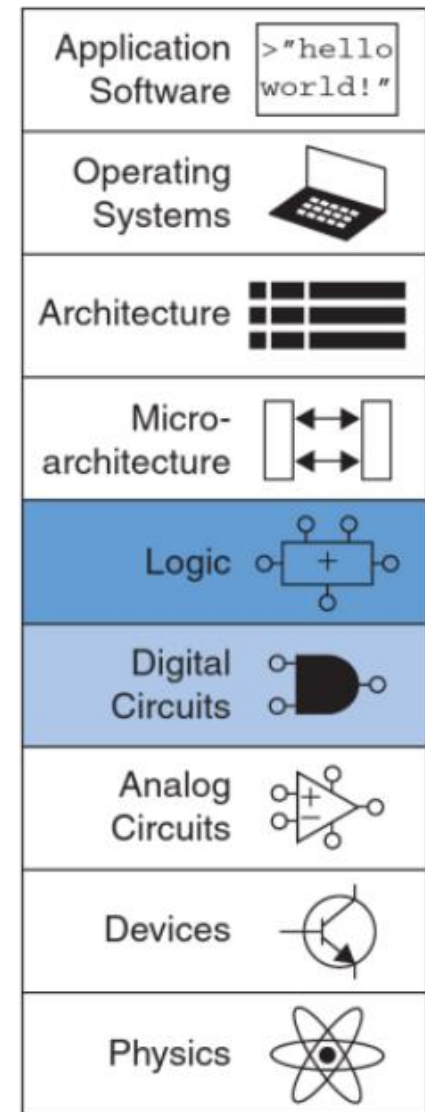
# Introductions: Students

- ❖ ~90 students registered, split across ECE and CSE
  - Different perspectives; can learn a lot from each other!
  - If you know others interested in adding, use the UW system as space becomes available (no add codes)
- ❖ Expected background
  - **Prereq:** EE271 or CSE369 – construction of synchronous digital systems (combinational + sequential logic), timing introduction, SystemVerilog introduction
  - **Prereq:** EE205 or EE215 – familiarity with circuits

# Course Motivation

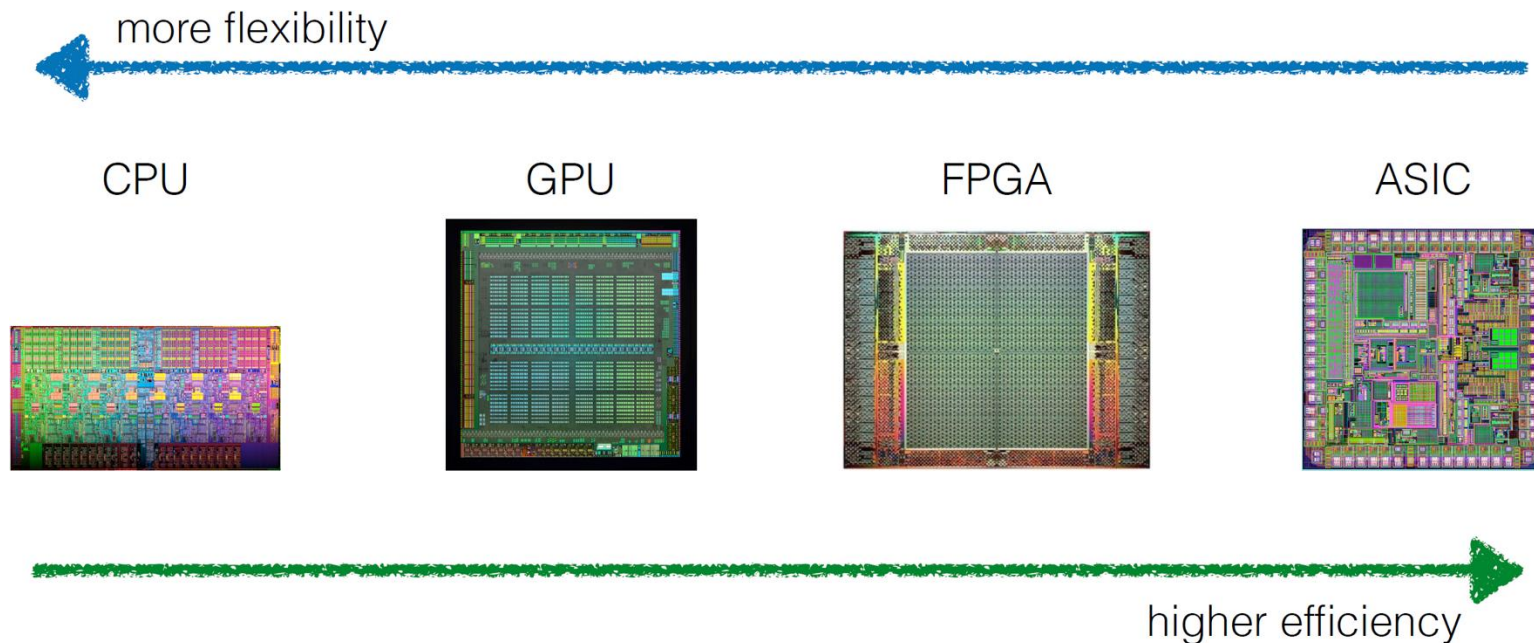
- ❖ More advanced digital logic design
  - Higher-level circuit design and analysis techniques
  - Interfacing with various devices/peripherals
  - How to implement algorithms *in hardware*
  - Practical timing analysis
  - “Verilog finishing school”

Harris and Harris. “Digital Design and Computer Architecture” 2<sup>nd</sup> ed.



# Course Motivation

- ❖ Heavy computational loads have put us in a golden age of **hardware specialization**
  - Tailor your chip architecture to the characteristics of a **stable** workload
  - More and more companies are building specialized chips



# Course Motivation

- ❖ Example: *FPGAs could replace GPUs in many deep learning applications*
  - <https://bdtechtalks.com/2020/11/09/fpga-vs-gpu-deep-learning/>
  - FPGAs have advantages in terms of power consumption, environmental robustness, and lifespan
  - “FPGA is a type of processor that can be customized after manufacturing, which makes it more efficient than generic processors.”
  - “But the problem with FPGAs is that they are very hard to program.” 😞

# Technical Communication

- ❖ Many of you are graduating soon!
  - Going off to do and build wonderful things
  - Try as you might, you can't really avoid society and other people
- ❖ It's no longer good enough to just get something working – you have to be able to explain it
  - Working in a group/team
  - Someone taking over as maintainer
  - Convince a client or venture capitalist or funding agency or students
- ❖ The lab reports and demos might seem tedious at first, but are valuable practice!



# Lecture Outline

- ❖ Course Introduction
- ❖ **Course Policies**
  - <https://courses.cs.washington.edu/courses/cse371/25sp/syllabus.html>
- ❖ Hardware Description Language
- ❖ SystemVerilog Review & Tips

# Course Meetings




## ❖ Lectures

- Lecture slides found on the website schedule
- Group work most lectures – with TAs available to help!

## ❖ Office Hours and Lab Demos

- Live demo using remote FPGA lab (LabsLand) and answering TA questions
  - Will support both in-person (ECE 007) and remote demos (Zoom)
  - When a TA is available, lab demo times are also office hours
- Office hours will use a Google Sheet queuing system (Tools → Office Hours Queue)

# Grading

- ❖ Labs (50%) 
  - 5 regular labs and 1 “final project”
  - Combination of lab reports, code submission, and lab demos
- ❖ Homework (20%) 
  - Combination of theoretical and coding questions
  - Interspersed between the labs
- ❖ Quizzes (30%) 
  - 5 roughly equally weighted quizzes, synced with homework
  - At end of lectures (exact dates subject to change)
  - Uploaded and graded in Gradescope
- ❖ “Straight-scale” grading scheme

# Collaboration Policy

- ❖ Complete your work as permitted (either individually or with partner/group)
  - Don't attempt to gain credit for something you didn't do and don't help others do so either
- ❖ **OK:**
  - Discussing/studying lectures and/or course material
  - *High-level* discussion of general approaches
  - Help with error messages, tools peculiarities, etc.
- ❖ **Not OK:**
  - Work in a larger group than permitted
  - Giving away solutions or having someone else (human or AI) do your assignment for you

# Collaboration Notes

- ❖ Make sure to add your groupmate(s) to your submissions in Gradescope
- ❖ Working in a group isn't a guarantee for success
  - Don't just split up the work: work simultaneously or at least communicate frequently (two brains are better than one!)
- ❖ We will provide GitLab repos for your convenience
- ❖ See the Collaboration page on the website (Course Info → Collaboration) for more info

# Deadlines

- ❖ Labs reports & code are submitted via Gradescope  
**Fridays before 11:59 pm**
  - Lateness is counted in *days* and cannot be submitted more than 2 days late (Sun 11:59 pm)
  - 6 late day tokens; 10% penalty for each late day after that
- ❖ 15-min lab demos within a week of report deadline
  - Usually in assigned slot (sign-up process)
- ❖ Homework cannot be submitted more than 1 day late
  - Solutions released to help you study for quizzes

Please talk to a staff member (preferably Vikram) about extenuating circumstances *as soon as they come up*

# To-do List

- ❖ Website:

- <https://courses.cs.washington.edu/courses/cse371/25sp/>

- Read over course policies and **HW/Lab Requirements PDF**
  - Fill out pre-course survey (due 4/4)

- ❖ Discussion: <https://edstem.org/us/courses/77539/>

- ❖ Install Quartus & ModelSim on your machine

- ❖ Register on LabsLand

- ❖ Look for partner(s) for homework and lab

- Figure this out *before* signing up for a lab demo slot

- ❖ Homework 1 & Lab 1 posted soon

# Lecture Outline

- ❖ Course Introduction
- ❖ Course Policies
- ❖ **Hardware Description Language**
- ❖ SystemVerilog Review & Tips



# Hardware Description Language (HDL)

- ❖ HDL is a specialized computer language used to describe the structure and behavior of digital logic circuits
  - Useful for rapid prototyping of digital hardware
- ❖ Comparison with programming language (*e.g.*, C)
  - Describes *hardware* instead of software
  - Intrinsically parallel instead of sequential
  - Includes explicit notion of time instead of just instructions
  - Compiled to target FPGAs and ASICs via netlists (more specific) instead of compiled to target CPUs via machine code (more general)

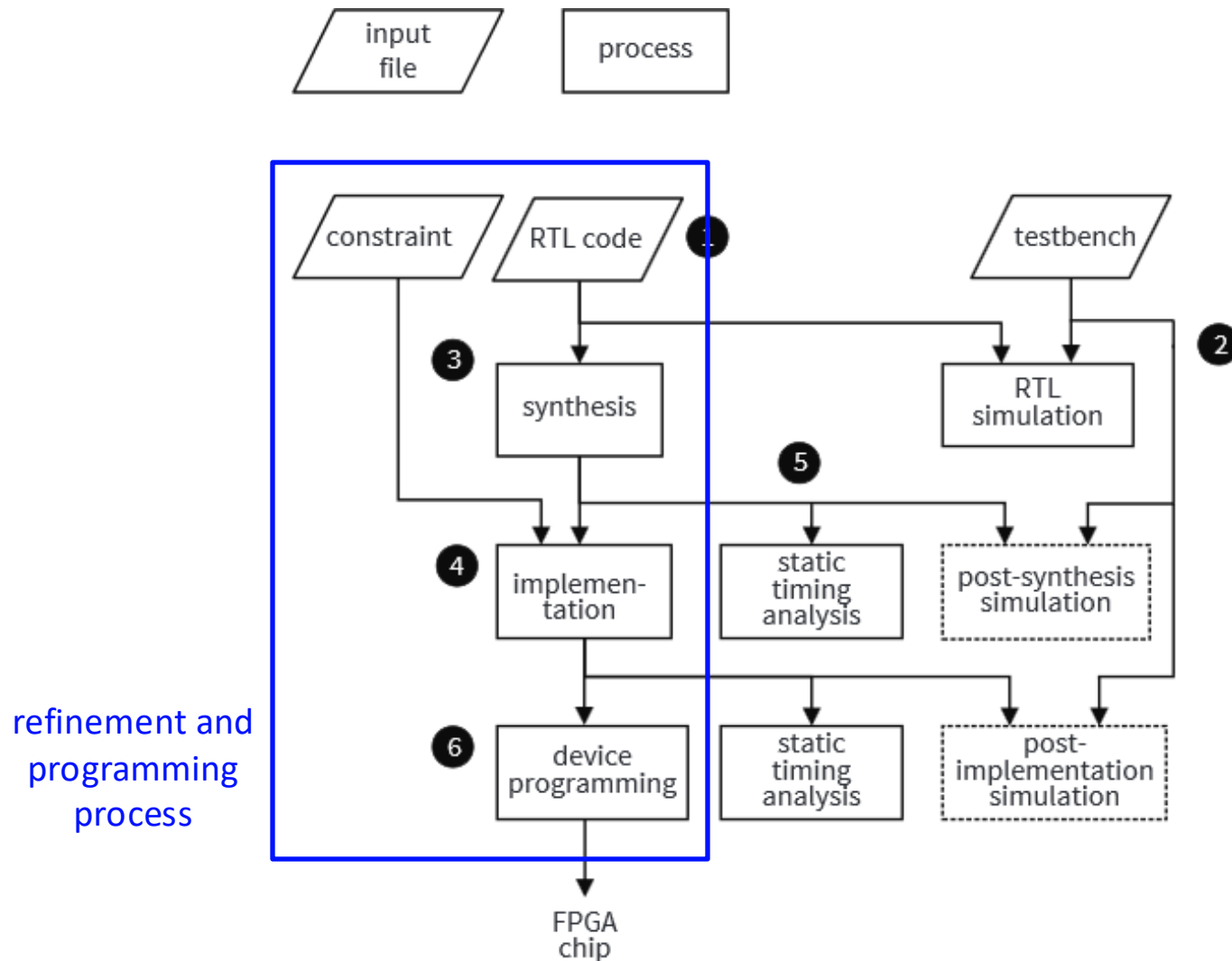
# Electronic Design Automation (EDA)

- ❖ EDA is a type of computer-aided design (CAD) specific to designing electronic systems
  - EDA tools allow for *synthesis* and *simulation*
- ❖ **Synthesis:** Compile HDL code into a netlist (*i.e.*, list of electronic components and wires that connect them)
- ❖ **Simulation:** Apply inputs to the simulated circuit so we can check output for correctness without involving or endangering hardware
  - Tracks the state of each structural element, handles the interactions between concurrent elements, and models the passage of time

# Design Flow Using Gates

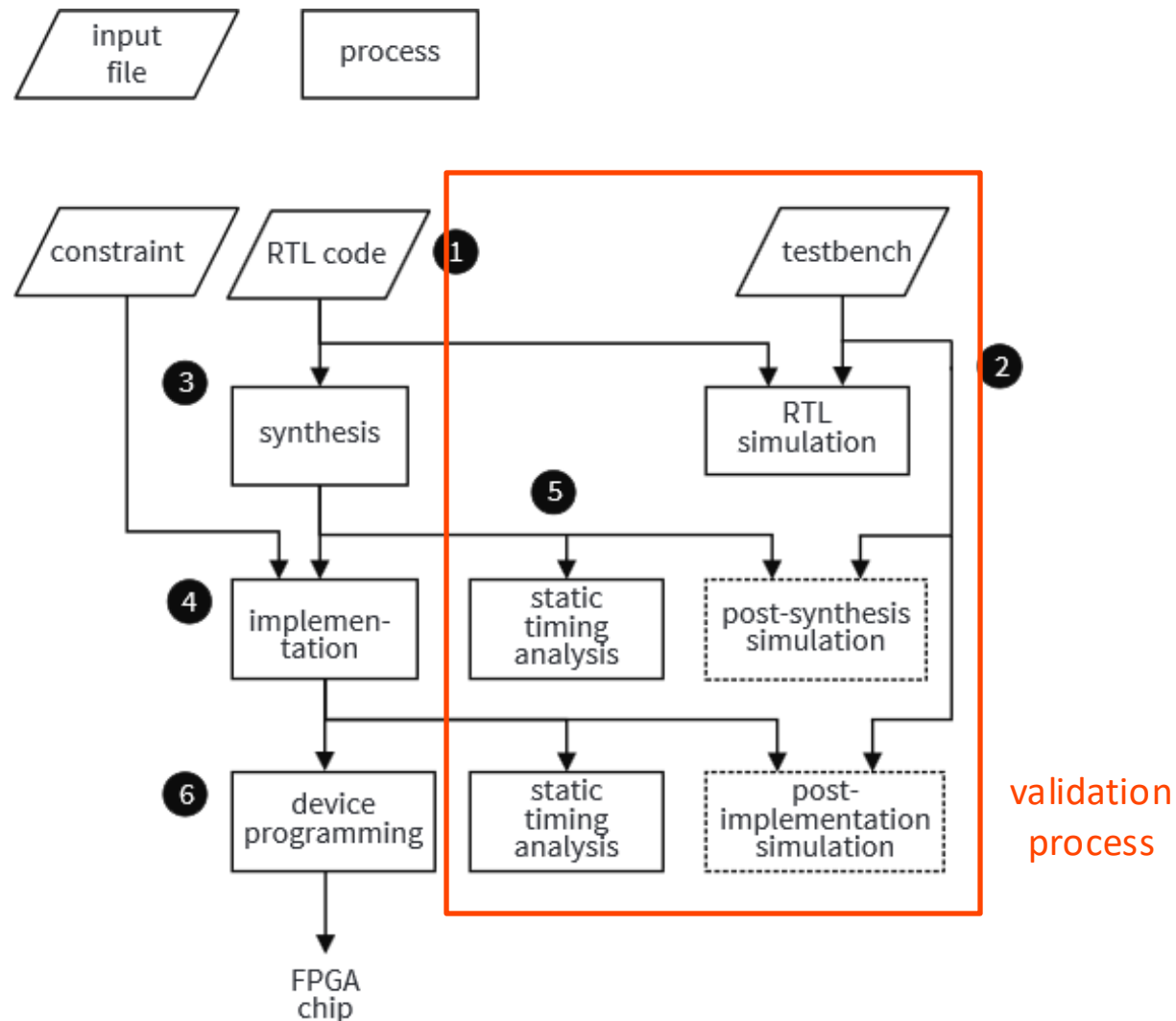
- ❖ Design of a synchronous digital system using gates:
  - 1) Write a specification
  - 2) If necessary, partition the design into smaller parts and write a specification for each part
  - 3) From the specification, draw a state machine diagram
    - Minimize the number of states when possible
  - 4) Assign a binary encoding to represent each state
  - 5) Derive the next state and output logic
    - Optimize the logic to minimize the number of gates needed
  - 6) Choose a suitable placement for the gates to optimize integrated circuit reuse and distance on printed circuit board
  - 7) Design the routing between the integrated circuits

# FPGA-Based Design Flow



Source: Figure 2.3 from "FPGA Prototyping" by P. Chu

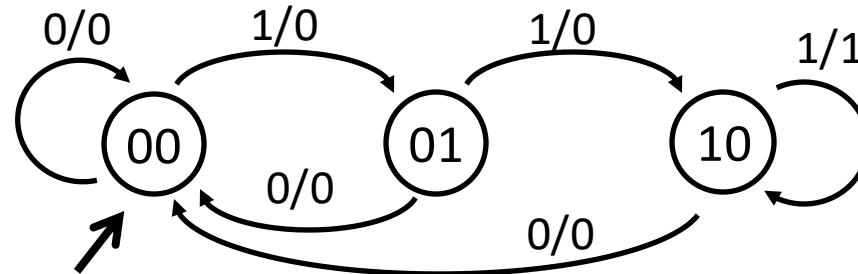
# FPGA-Based Design Flow



Source: Figure 2.3 from "FPGA Prototyping" by P. Chu

# Quartus/ModelSim/LabsLand Demo

## ❖ Using threeOnes FSM from 271/369



- Quartus/LabsLand and ModelSim have separate compilations
  - Clock divider is NOT FOR SIMULATION
  - Simulations can be run from a .do file or from individual ModelSim commands
  - Submodule signals can be added to waveforms!
  - Logical correctness can still lead to unexpected/unintuitive behaviors...
- ❖ For more, see “ModelSim Usage Guide” on website

# Technology Break

# Topic Outline

- ❖ Course Introduction
- ❖ Course Policies
- ❖ Hardware Description Language
- ❖ **SystemVerilog Review & Tips**
  - One of two leading HDLs along with VHDL-2019





# HDL Organization

- ❖ Most problems are best solved with multiple pieces – how to best organize your system and code?
- ❖ Everything is computed in parallel
  - We use routing elements to select between (or ignore) multiple outcomes/parts
  - This is why we use block diagrams and waveforms
- ❖ A module is not a *function*, it is closest to a *class*
  - Something that you ***instantiate***, not something that you ***call*** – hardware cannot appear and disappear spontaneously
  - Should treat modules as ***resource managers*** rather than temporary helpers
    - This can include having internal modules

# HDL Implementation & Testing

○ *Block diagram*

- 1) Create individual submodules
- 2) Create submodules test benches – test as usual
  - ■ CL – run through all input combinations
  - SL – take every transition that you care about
- 3) Create top-level module
  - Create instance of each submodule
  - Create wires/nets to connect signals between submodules, inputs, and outputs
- 4) Create top-level test bench
  - Goal is to check the interconnections between submodules – does input/state change in one submodule trigger the expected change in other submodules?

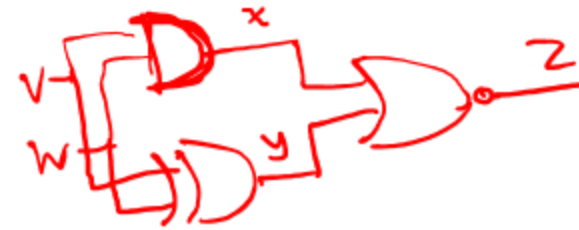
# Justin's SystemVerilog Tips

- ❖ Always plan out your design before you ever touch your keyboard
  - Block diagram naturally maps into module port lists
- ❖ Always think of the underlying hardware when you write your code
  - 1) **Structural:** lower-level modeling of gate-level connections
  - 2) **Behavioral:** higher-level modeling of logical behaviors
- ❖ Most modules are organized similarly, so develop and use coding patterns
- ❖ Comment your code and test thoroughly as you go
- ❖ Pay attention to compiler warnings and errors

# SystemVerilog Review Question

❖ Will the following compile? Is there a difference?

✓ ~~→~~ `logic v, w, x, y, z;`  
`and g1 (x, outv, inputw);`  
`xor g2 (y, v, w);`  
`nor g3 (z, x, y);`



✓ `logic v, w, x, y, z;`  
~~→~~ `nor g3 (z, x, y);`  
`and g1 (x, v, w);`  
`xor g2 (y, v, w);`  
~~✗~~ `and g1 (x, v, w);`  
`logic v, w, x, y, z;`  
~~✗~~ `xor g2 (y, v, w);`  
`nor g3 (z, x, y);`

# 271/369 SystemVerilog Refresher



## ❖ Combinational logic:

- Usually in `assign` or `always_comb` blocks and use blocking assignment (`=`)
- Implicit sensitivity list – updates anytime any input changes
- Testbenches for CL should just cycle through all input combinations at fixed time intervals (*e.g.*, `#20;`)

## ❖ Sequential logic:

- Usually in `always_ff` blocks and use non-blocking assignment (`<=`)
- Explicit sensitivity list – usually `@(posedge clk)`
- Testbenches for SL should take every transition that you care about, achieved by changing inputs every clock cycle



# Blocking vs. Nonblocking Assignment

- ❖ **Blocking** statement (=): the effects of these statements are “evaluated” sequentially
- ❖ **Nonblocking** statement (<=): the effects of these statements “evaluated” in parallel
- ❖ Exercise: draw out implied hardware

```
always_ff @ (posedge clk)
begin
    b  = a;
    c  = b;
end
```

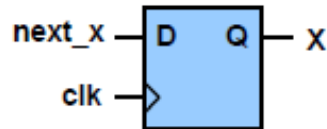
```
always_ff @ (posedge clk)
begin
    b <= a;
    c <= b;
end
```

Clarification from lecture: the slide is correct, my explanation was a little confusing. Blocking (=) get evaluated immediately, for non-blocking it happens at the end of cycle. See next page for hardware diagrams. Note that even on the left example we will still have a flip flop because this is an always\_ff block

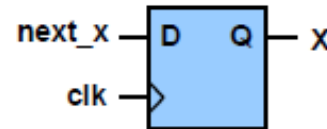
# Behavioral Non-Blocking Assignments

Note: this example is in Verilog, in System Verilog we would use `always_ff`

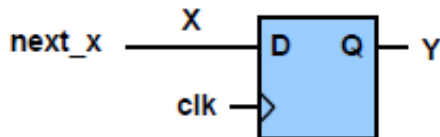
```
always @( posedge clk )
begin
    x = next_x;
end
```



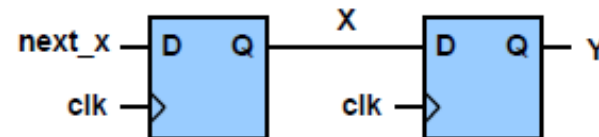
```
always @( posedge clk )
begin
    x <= next_x;
end
```



```
always @( posedge clk )
begin
    x = next_x;
    y = x;
end
```



```
always @( posedge clk )
begin
    x <= next_x;
    y <= x;
end
```



# 271/369 SystemVerilog Refresher



- ❖ We generally default to using `logic` for all signals, which is a 4-state data type that can be used as either a *net* or *variable*
  - Possible states (`logic` defaults to `X`):
    - `0` = zero, low, FALSE
    - `1` = one, high, TRUE
    - `X` = unknown, uninitialized, contention (conflict)
    - `Z` = floating (disconnected), high impedance
  - **Nets** (e.g., `wire`) transmit values and **variables** (e.g., `reg`) store data



# 271/369 SystemVerilog Refresher

- ❖ Multiple signals can be organized under the same name as a bus or array
  - A *bus*, also known as a *vector* or *packed array*, is a collection of a single data type
    - e.g., `logic [31:0] divided_clocks;`
  - “Regular” array syntax is known as an *unpacked array*
    - e.g., `logic an_unpacked_array[4:0];`
  - *Multidimensional arrays* can be combinations of packed and unpacked dimensions
    - e.g., `logic [3:0] two_D_array[4:0];`
    - Dimensions accessed left to right, starting with unpacked
  - Numbering matters (i.e.,  $\underbrace{[n-1:0]} \neq [0:n-1]$ )

# Review: Integers in Computing

- ❖ **Unsigned integers** follow the standard base 2 system
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + \dots + b_12^1 + b_02^0$
  - In  $n$  bits, represent integers 0 to  $2^n - 1$
- ❖ **Signed integers** use *Two's Complement representation*
  - $b_{w-1}$  has weight  $-2^{w-1}$ , other bits have usual weights  $+2^i$ 
  - In  $n$  bits, represent integers  $-2^{n-1}$  to  $2^{n-1} - 1$
  - Most significant bit acts as a *sign bit* (0 = pos, 1 = neg)
  - Handy negation procedure: take the bitwise complement and then add one ( $\sim x + 1 == -x$ )
- ❖  **The choice affects the behavior of operations such as bit extension, shifting, and comparisons**

# 271/369 SystemVerilog Refresher

We ended lecture 1 here since we were running out of time, leaving this here if you want to work through it

❖ Multi-bit constants: `<n> ' <s><b>#...#`

- `<n>` is width (*unsized* by default)
- `<s>` is signed designation (omit or 's')
- `<b>` is radix/base specifier (decimal by default)
- All letters are case-*insensitive*, `_` can be used to add spaces

Literal	Width	Base	Bits
3 'd6			
6 'o42			
8 'hAB			

Literal	Width	Base	Bits
42			
'b101			
-3 'd5			

- ❖ Compiler will usually warn you if there is a size mismatch
  - Can “cast” using `#' (<sig>)` syntax

# Basic Operators

## ❖ Possibly new:

- $371 \% 271 \rightarrow 100$  (remainder)
- $2 ** 3 \rightarrow 8$  (pow)

Type	Symbol	Description
Arithmetic	+	addition
	-	subtraction
	*	multiplication
	/	division
	%	modulus
	**	exponentiation
Shift	>>	logical right shift
	<<	logical left shift
	>>>	arithmetic right shift
	<<<	logical left shift
Relational	>	greater than
	<	less than
	>=	greater than or equal to
	<=	less than or equal to
Equality	==	equality
	!=	inequality
	===	case equality
	!==	case inequality
Bitwise	~	bitwise negation
	&	bitwise and
		bitwise or
	^	bitwise xor
Logical	!	logical negation
	&&	logical and
		logical or

# Ternary Operator

## ❖ Conditional assignment

- `select ? <then_expr> : <else_expr>`
  - If `select` is true, then evaluates to `<then_expr>`, otherwise evaluates to `<else_expr>`
- What does this look like in hardware?

## ❖ **Example:** tristate buffer

- `enable ? in : 'bZ`
  - When enabled, pass the input to the output, otherwise be high impedance

# Bit Manipulation

- ❖ Concatenation:  $\{sig, \dots, sig\}$ 
  - Ordering matters; result will have combined widths of all signals
- ❖ Replication operator:  $\{n\{m\}\}$ 
  - repeats value  $m$ ,  $n$  times
- ❖ Exercise: arithmetic right shift preserves the sign bit

```
logic [7:0] x = <some 8-bit constant>;  
// replicate the behavior of y = x >>> 3  
assign y =
```

# “Looping”

- ❖ Code is compiled to hardware, so no execution
  - “Loops” must be *statically unrolled* into multiple statements
  - Loops are just for convenience in code writing
- ❖ `repeat (#) <statement(s)>`
  - Makes # copies of statement(s)
- ❖ `for (i=0; i<#; i++) <statement(s)>`
  - Makes # copies of statement(s) that vary based on i
- ❖ `generate` (see reference docs)
  - More “powerful” for-loop typically used for:
    - 1) Module instantiation
    - 2) Changing the structure of parameterized modules
    - 3) Functional and formal verification using assertions

# Modules

- ❖ “Black boxes” that we define and instantiate that form the basic building blocks of our design hierarchy
  - **Ports** form the connections between a module and its environment
    - Ports have directionality (**input**, **output**, **inout**), which can be declared within the module or within the port list

```
module tristate(out, in, enable);  
    input logic in, enable;  
    output tri out;  
  
    assign out = enable ? in : 'Z;  
endmodule
```

```
module tristate(output tri out,  
                input logic in,  
                input logic enable);  
    assign out = enable ? in : 'Z;  
endmodule
```



# Module Instantiation

- ❖ Name an instance and define its port connections

- `<type> <name> (<port connections>);`

- ❖ Assume we have: `logic in, enable; tri out;`

## 1) Positional connections:

```
// must follow defined port ordering  
// signal names can be anything  
tristate my_tri(out, in, enable);
```

## 2) Named/explicit connections:

```
// any ordering & names allowed  
tristate my_tri(.out(out), .in(in), .enable(enable));
```

## 3) *.name* implicit connection:

```
// signal and port names must match exactly  
tristate my_tri(.out, .in, .enable);
```

# Parameters

## ❖ A **parameter** is a named constant

- Typically used for widths and timing

```
parameter N = 8;           // bus width
parameter period = 100;    // timing constant
```

## ❖ A parameterized module:

- `module <name> #(<parameter list>) (<port list>);`
- Parameters should be given default values
  - e.g., `#(parameter N = 8)`

## ❖ Extra exercises:

- Define a parameterized `tristate` (tristate buffer)
- Define a parameterized `multibitAND`

# Lab 1 Notes

- ❖ Read the spec carefully!
  - For scenarios that are not described, it's up for you to define; describe and defend your decisions in your report
  - Also read *371\_Assignments.pdf*
- ❖ Plan and design *before* you start coding!
- ❖ Test your code in small pieces *as you go*
  - Lab report due before your demo
  - Short sessions (3 min) on LabsLand

