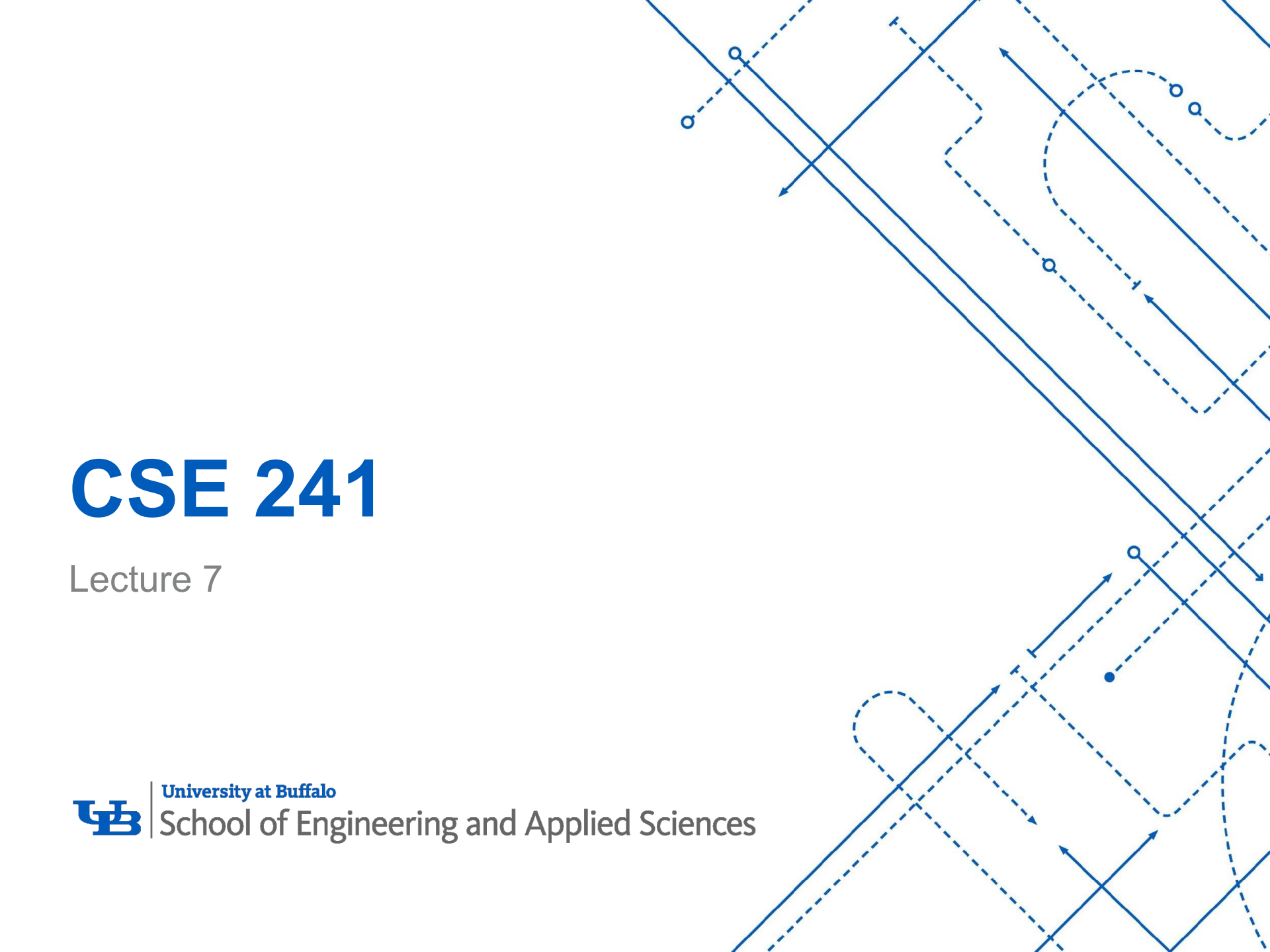


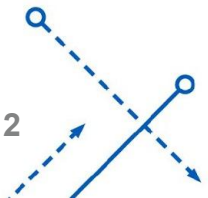
CSE 241

Lecture 7

University at Buffalo
School of Engineering and Applied Sciences

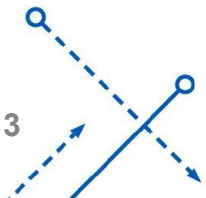
Overview for this lecture

- Reminders
- Sequential Logic
- Introduce Verilog



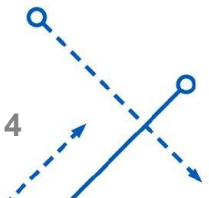
Reminders/Announcements

- How About 5 Bonus Points on the Final?
  - Reach 70% class participation on the course evaluation
  - As of 6/24 you were at 10%
- 2 More HW
  - Today's and One Assigned Wednesday



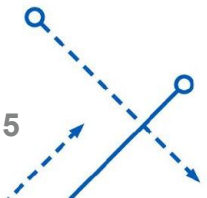
Lab Wednesday/Monday- Last Lab

- Doing 1 big lab instead of 2 smaller labs
- Have not checked to see if the X11 software is there
- Can not program the FPGA from the machine
  - Will need to set up other times for demonstration
    - Due to the holiday this may mean people will be doing demonstrations after the final exam
  - If you have Vivado on your personal machine you can run the demonstration from there

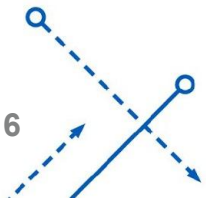


The Plan

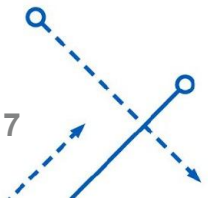
- Wednesday- Verilog
- Monday- Review and Practice Exam
- Next Wednesday- Final Exam



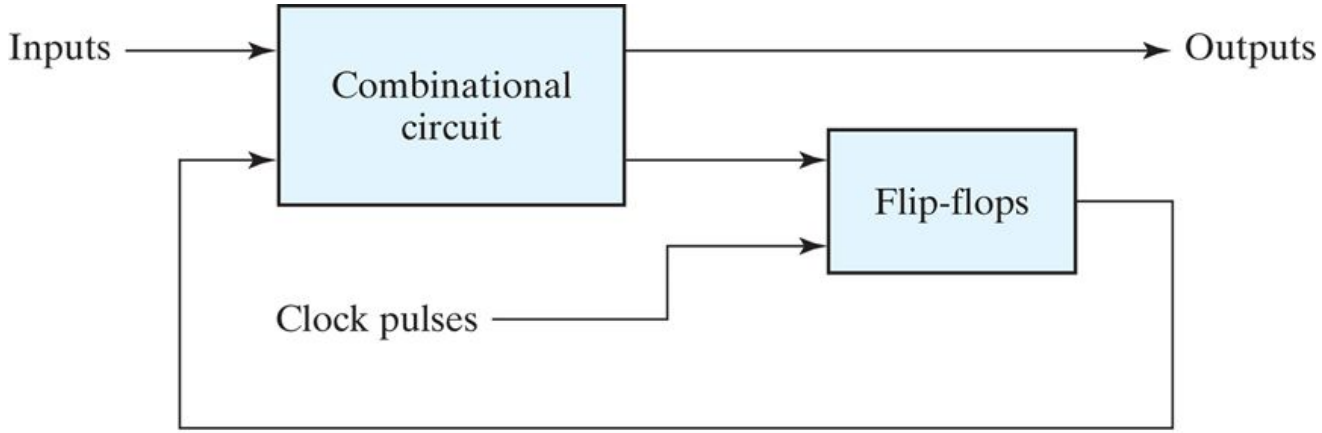
Questions?



Sequential Logic Wrap Up



What Sequential Circuits Look Like

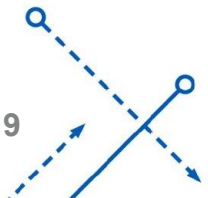


(a) Block diagram

(b) Timing diagram of clock pulses

Types of Sequential Circuits

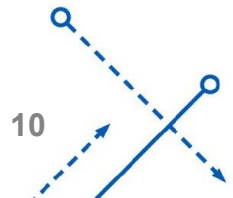
- Moore
  - outputs depend only on FF states
  - no direct dependence on circuit inputs W
  - simpler to design
  - usually requires one or more extra states
- Mealy
  - outputs depend on FF states and circuit inputs W



Sequential Circuit Design Steps

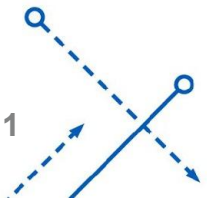
1. Construct a state diagram.
2. Convert the state diagram to a state table.
3. Minimize the state table.
4. Assign flip-flop values to the states.
5. Construct a transition table.
6. Choose a desired FF type.
7. Determine the FF input and circuit output equations.
8. Draw the circuit diagram.

(We will usually not do this.)



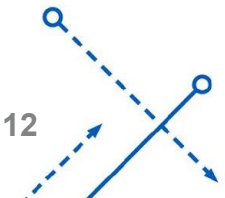
More on Moore and Mealy

- How can you tell if a circuit is Moore or Mealy?
- Moore circuits have outputs that depend only on flip-flop states, not directly on circuit inputs.
  - The state diagram has the output in the state circle.
  - The state table has a special column for output.
- Mealy circuits have outputs that depend on flip-flop states and circuit inputs.
  - The state diagram shows the output on the transition arrow.
  - The state table shows outputs in input columns.



Implementing with something other than D-FlipFlop

We will consider T-FlipFlop and JK FlipFlop



For JK and T

1. Determine the flipflop input equations in terms of present state and input variables
2. List the binary values for each input equation
3. Use the flipflop characteristic table to determine the next state values

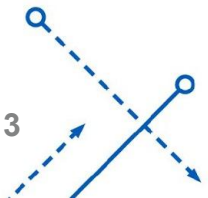


Table 5.1
Flip-Flop Characteristic Tables

JK Flip-Flop

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

D Flip-Flop

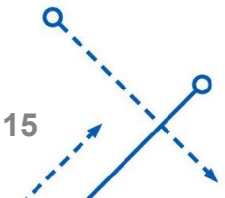
D	Q(t + 1)	
0	0	Reset
1	1	Set

T Flip-Flop

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Let's Do An Example

Arbitrary Counting



Asynchronous Circuits- A High Level Quick Cover

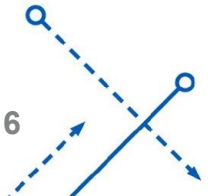
Why?

It is part of sequential circuits and it really is just an extension of the base concepts so you need to have some awareness of it

So how much Asynchronous will be on the exam?

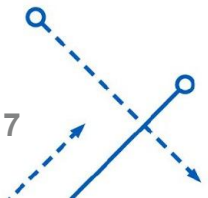
You need to know what the difference between synchronous and asynchronous is, including how to recognize the difference

BTW- It's not in the book



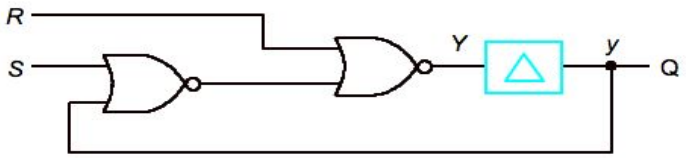
Asynchronous Behaviour

- Some asynchronous basics:
  - There is no clock pulse to synchronize the circuits here.
  - There are no flip-flops for state variables.
  - To get reliable behavior, circuits must receive their inputs in a nice, controlled manner.
  - We'll assume just one input changes at a time.
  - Moreover, input changes must occur respecting propagation delay so that the circuit is stable before another input arrives.
  - Circuits adhering to these constraints are operating in fundamental mode.



An Asynchronous SR Latch

- The little Δ indicates an arbitrary time delay.
  - The NOR gates are ideal and have no delay.
- Y is the next state, y is the present state.
- Whenever Y = y, the circuit is stable.
  - Stable states are indicated by the little circle.
- Consider the input sequence SR = 00, 01, 11, 01. When does the circuit stabilize?



(b) Circuit with modeled gate delay

Present state y	Next state SR = 00	01	10	11
	Y	Y	Y	Y
0	(0)	(0)	1	(0)
1	(1)	0	(1)	0

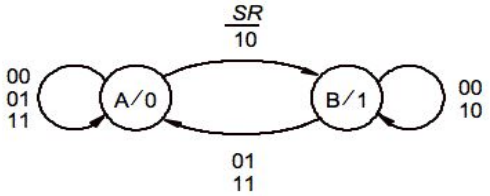
(b) State-assigned table

Turning that into an FSM

- Going backwards from circuit to state machine.
- Let A be when y = 0 and B be when y = 1.
- Notice how we've notated the two inputs to the state machine.
- One would be led to believe synchronous and asynchronous circuits have the same design path; they don't.
  - Asynchronous circuits are much harder.

Present state	Next state				Output Q
	SR = 00	01	10	11	
A	(A)	(A)	B	(A)	0
B	(B)	A	(B)	A	1

(a) State table

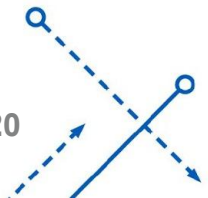


(b) State diagram

Figure 9.2. FSM model for the SR latch.

Some More Terminology

- One the previous slides, the terminology from our synchronous circuits was used, but that's not common. Perform the following substitutions:
  - State Table = Flow Table
    - We flow from one state to another.
  - Transition Table = Excitation Table
    - Depicts transitions of state variable by exciting the next-state variables.



Gated D Latch Redux

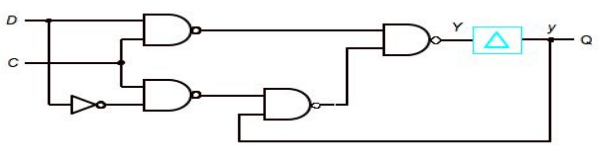
- Let's redo a gated D latch, treating the clock as just another random input.
- Let's build it up. Shown at right is the excitation table.
  - Remember, C = Clock, D = Data.
- The minimal function is: Y=CD+C'y
- The best function is: Y=CD+C'+Dy
  - It's better because it's hazard-free.

Present state y	Next state CD = 00	01	10	11	Q
	Y	Y	Y	Y	
0	(0)	(0)	(0)	1	0
1	(1)	(1)	0	(1)	1

(b) Excitation table

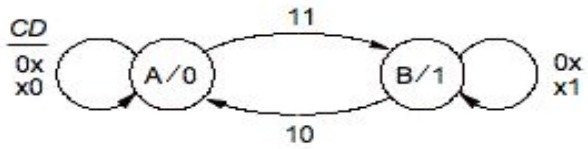
More on that D Latch

Here's the circuit, flow table and state diagram for the D latch.



Present state	Next state CD = 00	01	10	11	Q
A	(A)	(A)	(A)	B	0
B	(B)	(B)	A	(B)	1

(c) Flow table



(d) State diagram

D Flip-Flop Asynchronously

- This is the master-slave flip-flop.
- Y_m and y_m are the state of the master, Y_s and y_s are the state of the slave.
- C and D are the same as before.
- Let's walk through this excitation table and try to follow it.

Present state $y_m\ y_s$	Next state CD = 00	01	10	11	Output Q
	$Y_m\ Y_s$				
00	(00)	(00)	(00)	10	0
01	00	00	(01)	11	1
10	11	11	00	(10)	0
11	(11)	(11)	01	(11)	1

(a) Excitation table

D Flip-Flop Flow Tables

- The top flow table is a pretty direct translation of the excitation table.
  - But consider inputs CD = 01 in state S1 and CD = 00 in state S3. Are these meaningful?
  - No.
- The bottom flow table removes entries that can't happen because only one input can change at a time.

Present state	Next state				Output Q
	CD = 00	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	S1	(S2)	S4	1
S3	S4	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(b) Flow table

Present state	Next state				Output Q
	CD = 00	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	–	(S2)	S4	1
S3	–	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(c) Flow Table with unspecified entries

State Diagram

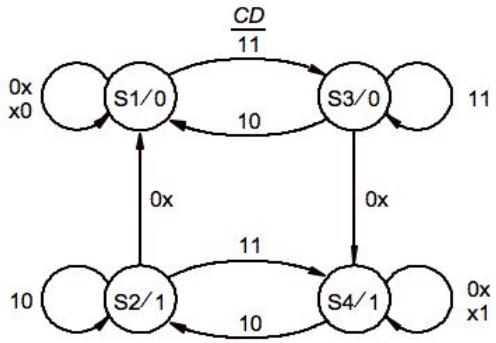


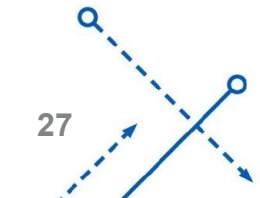
Figure 9.7. State diagram for the master-slave D flip-flop.

Synthesis of Asynchronous Circuits

- Devise a state diagram from the textual description of the required behavior.
- Derive the flow table.
- Reduce states if possible.
- Perform state assignment.
- Derive the excitation table.
- Obtain the next-state and output expressions.
- Draw the circuit (optional).

What makes it hard?

- When creating the diagram (or flow table), we have to make sure the circuit is stable before we try to output anything.
- Minimization is non-trivial.
- State assignments are not just done with the intention of minimizing the combinational logic.
  - We have to avoid unreliable state assignments.



Hazards

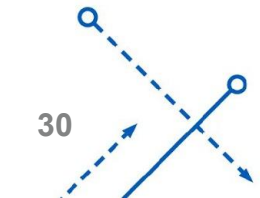
- Hazards can be generally lumped into two categories:
  - Static Hazards: The output is supposed to stay at one value, but doesn't.
  - Dynamic Hazards: The output is supposed to transition, but oscillates briefly before settling down.
- Check out the pretty pictures on the next slide!

(a) Static hazard

(b) Dynamic hazard

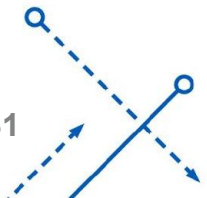
Static Hazards

- Consider the function
  - $f(x_1, x_2, x_3) = \Sigma(1, 3, 6, 7)$.
- Let's do the K-map and draw the resulting circuit.
- Then, consider what happens when the circuit is in the state $x_1 = x_2 = x_3 = 1$ and x_1 transitions to a 0.
- How can we fix this?



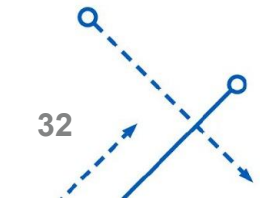
Detecting and Eliminating Static Hazards

- A static hazard exists whenever two adjacent ones in a K-map are not covered by a prime implicant in the resulting minimal function.
- To fix it, include additional prime implicants whenever there are two adjacent ones.
- The function won't be minimal anymore, but there is no undesirable behavior.



Why do we care?

Because hazards can cause undesirable, problematic or even dangerous behavior in our state machines, because the state machine will enter the wrong state.

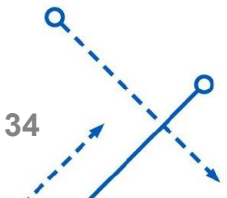


What is next?

VERILOG!

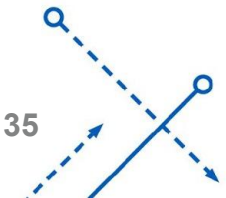
What is Verilog?

- Hardware description Language (HDL)
  - One of the classes of it anyway
- Allows for design, synthesis, and testing of larger circuits
- Developed in 1984, Resembles C
- Now IEEE 1364
  - Verilog95 and Verilog01
  - Alternative is the DoD-approved VHDL
  - Another newer challenger: System C
  - There is a new Verilog variant called “SystemVerilog” too.



Key Terminology
-slightly different from the programming you know

- Netlist- the list of physical components and their interconnects
- Module- a function
- Timing verification- timing simulation that confirms performance at specific operating speeds
- Portlist- the list of the variables that are going in and out of the module
- Primitives- built in base functions and keywords to use
  - these can not be used as module names
- Instantiate- instance of a module or primitive call



General Syntax

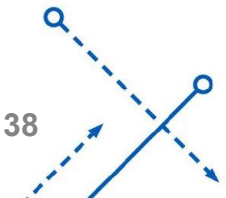
```
module module_name(portlist);

//body of the module


endmodule
```

What goes in the module

All the stuff to design your circuit behaviour

3 Styles

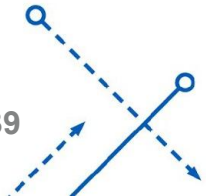
1. Gate Level
    a. Uses primitives to implement everything
    b. Can have multiple instantiations of the primitives
    c. Sometimes called structural
2. Continuous Assignment
    a. Uses the assign statement, once the statement is ran any change to part of the expression will update the assignment
3. Behavioural
    a. Uses the keyword always, it is a procedural approach



Let's look at a Verilog Program

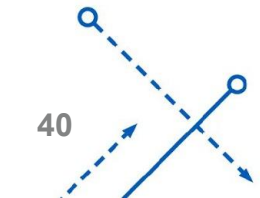
```
module example1(x1, x2, x3, f);
    input x1, x2, x3;
    output f;

    and(g, x1, x2);
    not(k, x2);
    and(h, k, x3);
    or(f, g, h);
endmodule
```

- Implements function
  - f = (x1 • x2) + (~x2 • x3)
- Is a gate level design
  - Structural design is easier, but more time consuming
  - Is not really appropriate for large designs
- Note that we need intermediate values to "force" sequential execution.



What primitives do we have?

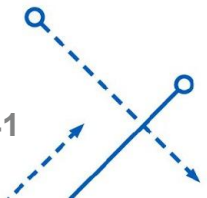
- There are 12 basic gates predefined for us:
  - **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf** and 4 three state types
- Notice: case matters!



Let's look at that in Continuous Assignment Form

This approach is also known as data flow

```
module example3(x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

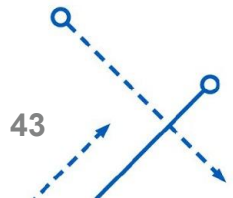



Now Behavioural

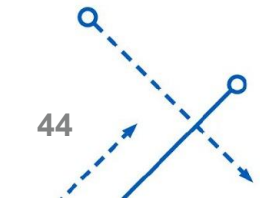
```
module example5(x1, x2, x3, f);
        input x1, x2, x3;
        output f;
        reg f;
        always @(x1 or x2 or x3)
        begin
                if (x2 == 1)
                        f = x1;
                else
                        f = x3;
        end
endmodule
```

Some Comments

- Comments
  - Single line: //
  - Multi-line: /* … */
- begin and end
  - Replace { and }, respectively
- Code in an always block is executed sequentially.
- Verilog compilers are not nice.

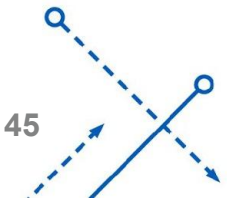


Let's Practice Some with Some Examples and Exercises



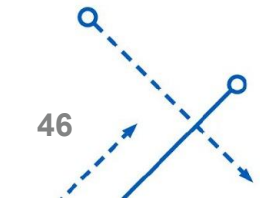
The *wire* Data Type

- For internal connections with no connection to an input or output port, we use the wire data type.
- Don't let the word "wire" throw you off. This is not just a conductor.
  - Can drive loads!
- Example:
  - wire [3:1] C;
  - A 3-bit wide internal signal



The *reg* Data Type

- Variable that can hold it's value
- Does not need a driver
  - Can update any time values are assigned

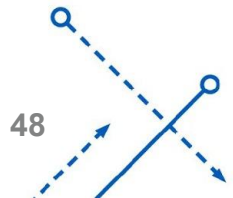


The *parameter* Keyword

- The parameter keyword is like a #define in C/C++. It lets you define a constant.
- Example:
  - parameter n = 4;
    wire [n-1:0] W;
    - W is then a 4-bit wide signal.
    - Some design houses don't use this.

Verilog *for* statements

- The Verilog for is almost identical to a C/C++/Java for except:
  - The unary ++ and -- operators don't exist.
  - Therefore, you must do something like: k = k + 1
  - The { and } are replaced by begin and end, respectively.
- Like the "if" statement, a "for" can only exist in the sequential always or initial block.

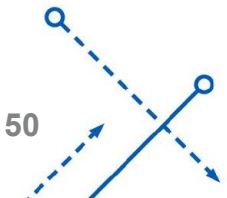


Number Representation in Verilog

- <bits>'<radix><digits>
- Examples:
  - 4'b1101
  - 8'd66
  - 12'o7734
  - 12'hBA9
- Experience suggest it is always worth being specific in data sizes.
- Even things like mySignal = 16'd0;

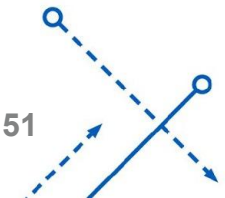
Conditional Operator

- Also called ternary operator
  - conditional ? true_expr : false_expr
  - Example:
    A = (B < C) ? (A+5):(A+2);
  - Don’t use this thing.
    - Seriously.



The *case* Statement

```
case(expression)
        alt1: stuff;
        alt2: stuff;
        … etc. …
        [default: stuff;]
endcase
```

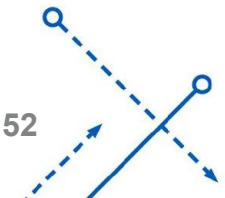


Why did we just spend that time on Instantiation?

Modular design is our friend!

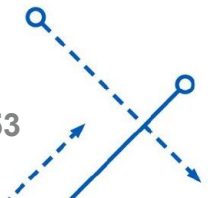
How do we do it?

Make a second module and call it from the first one.



Instantiation

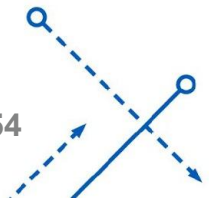
- Module name, followed by a unique instance name and a port map phrase.
  - Example:
    - fulladd stage0 (carryin,x0,y0,s0,c1);
    - Port map
  - Connects signals from here to there
- Same module may be instantiated more than once, but with a unique instance name.
- The module doing the instantiation must know what the definition of the module is.
- They either need to be in the same file or in the same project (depending on the complexity of your environment).



Tasks and Function

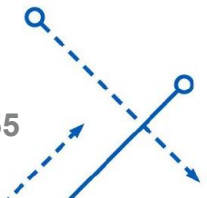
Recall How I said a module was like a function, starting out that is true but there are Tasks and Functions in Verilog too

- Both act like methods in Java
- Allow you to create some block of code that gets called an arbitrary number of times.



Key Differences- Task VS Function

- A function shall execute in one simulation time unit; A task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks or functions.
- A function shall have at least one input type argument and shall not have an output or inout type argument; A task can have zero or more arguments of any type.
- A function shall return a single value; A task shall not return a value.



Highlights

- **wire** and **reg** are used internally
- Port lists are inputs and outputs and if using a module you need to use correct order
- 4 Schools: Gate (structural), Behavioral (think always statements), Data Flow (continuous assignment), and CMOS (same principle as Gate but even lower level, we didn't talk about that one)
- modular design is your friend (or copy/paste but you should know better then to waste space and risk errors with that approach)

Once you have the program, what do you do with it?

- This will be the focus on Wednesday- plus more examples and concepts
- You program a FPGA- Field Programmable Gate Array or your simulate your circuit
- Why- It is a fast and easy way to test out your circuit
- How- Use Vivado- this is our IDE/Simulator and Programmer Suit
  - You can download it for free- huge space hog
  - On Timberlake- we will walk through how to get there
    - If you have a PC you will need a software for X11 forwarding: https://www.buffalo.edu/ubit/service-guides/software/downloading/windows-software/managing-your-software/x-win32.html