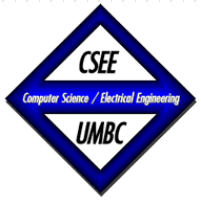


University of Maryland Baltimore County
Department of Computer Science and Electrical Engineering

CMPE 212L, Principles of Digital Design Laboratory

Instructor: Prof. Younis

TA: Ahmed Shahin, Ali Ahmed, Kevin Bochinski

Email: ashahin1@umbc.edu

Discussion 6 Goals

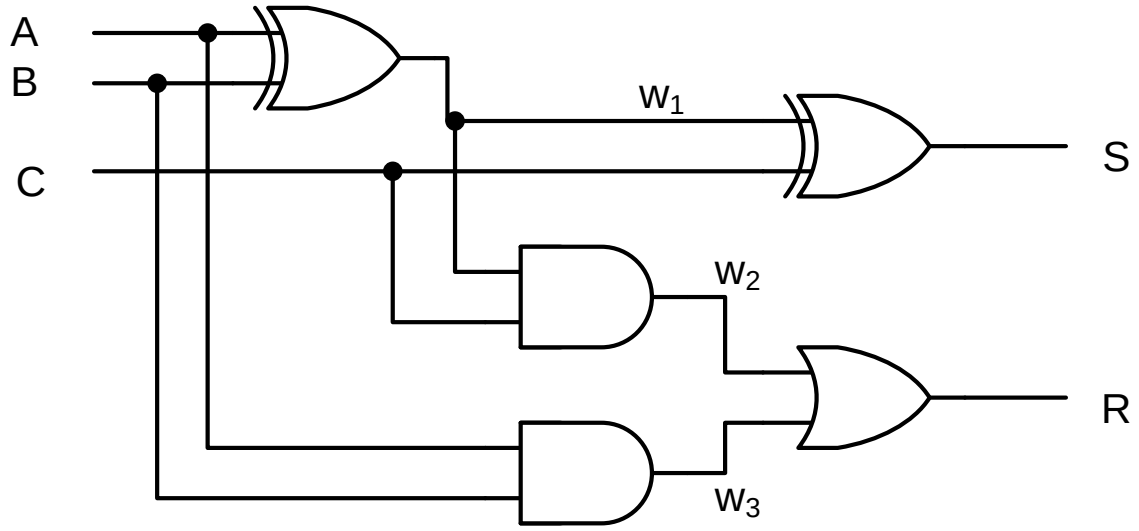
- Lab 6
  - Description of lab
  - Running and checking your results
- Learn more Verilog
- Project 1 Description
- Writing Verilog Code
  - Verilog on Windows
  - Verilog on The GL Account

Lab 6 Description

- This lab requires **Verilog coding ONLY**, and does not require you to build a circuit.

Lab 6 Description

- Write **two** Verilog files and run them for the TA's:
  1. A module which describes the structure of the **circuit below**.
  2. A testbench which tests the functionality of the module.



Running and checking your code

- There should be no errors, with truth table output:
- If this is your **first** time running Verilog, then refer to the slides at the end of this discussion.

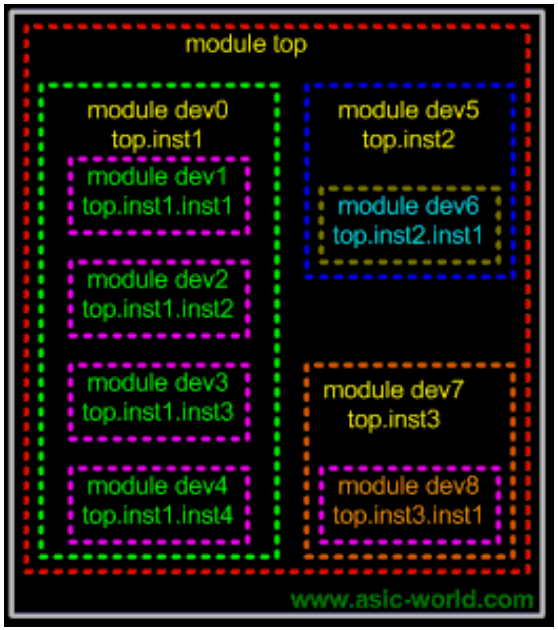
A	B	C		S	R
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

Verilog Review

- Verilog is a Hardware Description Language.
- Modular implementation of circuit components:
  - Modules form the building blocks of circuit components.
  - Bigger modules are made up of smaller modules.
- 2 different ways to program ("Levels of Abstraction"):
  - Behavioral – no direct mapping to circuit elements (if-statements, loops).
  - Structural – represents circuit components (gates, lower level modules).

Modules

- Most **basic unit** of hierarchy in Verilog
- Can be a **single circuit element**, or a **collection** of lower level modules connected together.
- Contained in 1 Verilog file.
- Multiple modules can reside in 1 file, but this is **not** recommended.
- Module name should **match** the file name. So the module 'adder' would reside in the file named 'adder.v'.



Modules continued

- Defining a module:

module *<module_name>*(*<module_terminal_list>*);
...
<module internals>
...
endmodule

- Defining a Half Adder module:

module HalfAdder(A, B, S, Cout);
...
<functionality of Half-Adder>
...
endmodule

- We can describe the functionality of a module with either Structural code or Behavioral code.

2 Levels of Abstraction

Structural: Gate Primitives

- Predefined in Verilog. Can have multiple inputs.

```
and and_1 (out, in0, in1);

nand nand_1 (out, in0, in1,in2);

or or_1 (out, in0, in1);

nor nor_1 (out, in0, in1, in2, in3);

xor xor_1 (out, in0, in1, in2);

xnor xnor_1 (out, in0, in1);
```

Behavioral: Operators

- Bitwise: ~, &, |, ^
- Logical: !, &&, ||
- Reduction: &, |, ^
- Arithmetic: +, -, *, /, **
- Relational: >, >=, ==, !=
- Shift: >>, <<

Integer Literals

- Used for as an **easier** and **more explicit** way to assign values, in comparison to: `A = 1001  //want to be binary (-7)`$_{16}$

- Specified as follows: **<size>'<base><value>**
  - **<size>** is optional. Determines the total number of bits represented by the value. If not given, then the default size is 32 bits.
  - **<base>** is required. Specifies whether the value is in binary (`b`), octal (`O`), decimal (`d`), or hex (`h`).
  - **<value>** is required. Specifies the integer value. If you want the value to be negative, then put a negative sign in front of everything.
- Example: `A = -4'h7` allocates 4 bits and stores the binary representation of $(-7)_{16}$.

Initial, Always, and Always@ blocks

- Initial block is executed **once** at simulation time = 0.
- Always blocks **loop** to execute over and over.
  - Always (no '@') will also begin execution at time = 0, and repeat as soon as it's done with all statements in the block.
  - Always@(event) will wait until the event specified before executing. It won't repeat until the event specified occurs again.

System Tasks and Functions

- Used to **help debug** your code.
- NEVER used as part of your circuit structure, and usually found in testbenches.
- Names begin with a '`$`'.
- `$display`, `$strobe`, `$monitor` - print strings
- `$time` - simulation time
- `$reset`, `$finish` - program execution
- `$random` - random integer
- `$fopen`, `$fdisplay`,`$fmonitor`,`$fwrite` -write to files

$display, $strobe, $monitor

- These functions **display text** on the screen during simulation.
- `$display` and `$strobe` print once each time they are executed.
  - `$display` will print at the time of its execution
  - `$strobe` will wait until the simulation finishes running to print
  - Example: `$display("Value of x = %b", x);`
  - Format characters include `%d` (decimal), `%h` (hex), `%b` (binary), `%c` (character), and `%s` (string).
- `$monitor` prints every time one of its parameters changes.
  - Example: `$monitor("Value of x = %b", x);` will print the string each time that `x` changes value.

$time, $random, $reset, $finish

- `$time` returns the current simulation time, and can be used within the `display/strobe/monitor` functions.
- Example: `$monitor($time, " Value of x = %b", x);` will print the string when `x` changes, as well as the simulation time at which `x` changes.
- `$random` generates a random integer every time it's called.
- `$reset` sets the simulation time back to 0.
- `$finish` exits the simulator and returns control back to the operating system.

$fopen, $fdisplay, $fwrite, $fmonitor

- These functions **write to files**.
- `$fopen` opens an output file and gives the open file a handle for use by the other commands.
- `$fclose` closes the file and lets other programs access it.
- `$fdisplay` and `$fwrite` write formatted data to a file whenever they are executed. `$fdisplay` inserts a new line after every execution but `$fwrite` does not.
- `$fmonitor` writes to a file whenever any of its arguments changes.

Blocking / Non-blocking assignments

- There are **two ways to make assignments** in Verilog, so far we have only used the first: Blocking and Non-blocking assignments.
- Blocking assignments
  - Use the `'='` token
  - Occur **sequentially**
- Non-blocking assignments
  - Use the `'<='` token
  - Occur **concurrently** (in parallel)

Blocking / Non-blocking assignments

Blocking

- Assignments are evaluated and assigned in ONE step.
- Execution flow within the simulation is "**blocked**" until an assignment is complete. This forces sequential execution.

```
word[15:8]=word[7:0];
```

Non-blocking

- Assignments are evaluated and assigned in TWO steps:
  1. Right side of assignments are evaluated first.
  2. Assignments to left side are postponed until other evaluations in the current time step are completed.

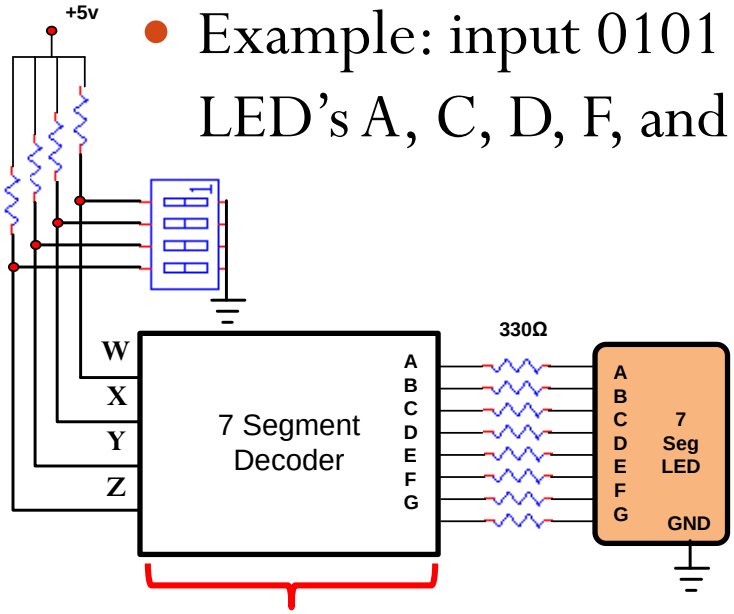
```
word[15:8]<=word[7:0];
```

Verilog Guides

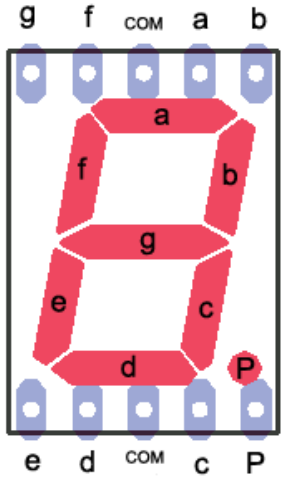
- Free online course: http://vol.verilog.com/
- Free online tutorial: http://www.ece.umd.edu/class/enee359a.S2008/verilog_tutorial.pdf
- Verilog Quick reference: http://www.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf

Project 1 Description

- **Build a Decoder** which will take a 4-digit binary number and display the corresponding base-10 digit.
- Inputs – **4** binary digits: WXYZ
- Outputs – **7** LED controls: A, B, C, D, E, F, and G.
- Example: input 0101 results in a 5 being displayed, or LED's A, C, D, F, and G turn on.



You make this part!



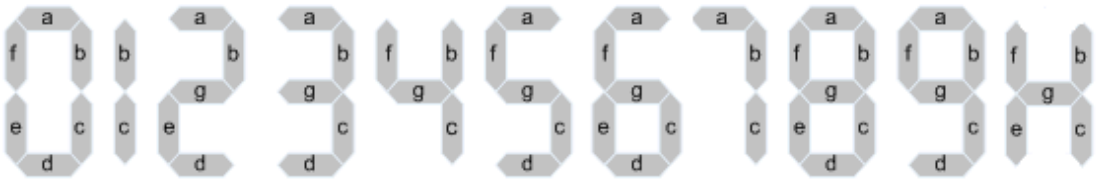
Courtesy: Erin DeLong

Truth Table for a 7-Segment Display

Truth Table for a 7-segment display

Individual Segments							Display
a	b	c	d	e	f	g	
×	×	×	×	×	×		0
	×	×					1
×	×		×	×		×	2
×	×	×	×			×	3
	×	×			×	×	4
×		×	×		×	×	5
×		×	×	×	×	×	6
×	×	×					7

Individual Segments							Display
a	b	c	d	e	f	g	
×	×	×	×	×	×	×	8
×	×	×	×		×	×	9
	×	×		×	×	×	H

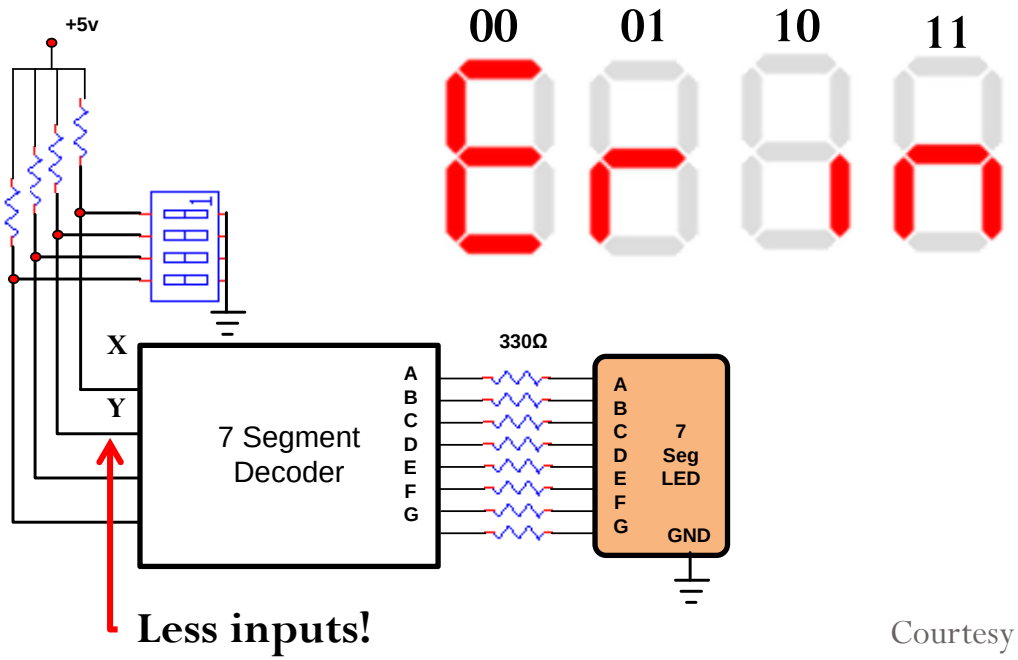


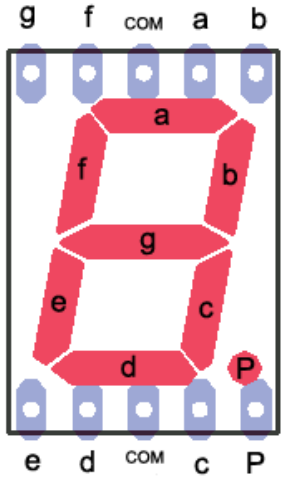
7-Segment Display Elements for all Numbers.

Courtesy:Erin DeLong
CMPE 212L Principles of Digital Design Laboratory

Easier Example Problem

- Suppose that we have:
  - **2 inputs**, X and Y, instead of 4 inputs.
  - Still 7 outputs, A, B, C, D, E, F, and G.
  - We want to display the following letters for these inputs:





Courtesy: Erin DeLong

Module for Easier Example

```
1  module easyDecoder(a,b,c,d,e,f,g,X,Y);
2
3      input X,Y;
4      output a,b,c,d,e,f,g;
5
6      wire m0,m1,m2,m3;
7
8      and(m0,~X,~Y);   // a output
9      or(a,m0,0);
10
11     or(b,0,0);       // b output
12
13     and(m2,X,~Y);    // c output
14     and(m3,X,Y);
15     or(c,m2,m3);
16
17     or(d,m0);        // d output
18
19     and(m1,~X,Y);    // e output
20     or(e,m0,m1,m3);
21
22     or(f,m0);        // f output
23
24     or(g,m0,m1,m3); // g output
25
26 endmodule
```

- 'easyDecoder.v'

$$f_a(X,Y) = \bar{X}\bar{Y}$$

$$f_b(X,Y) = 0$$

$$f_c(X,Y) = X\bar{Y} + XY$$

$$f_d(X,Y) = \bar{X}\bar{Y}$$

$$f_e(X,Y) = \bar{X}\bar{Y} + \bar{X}Y + XY$$

$$f_f(X,Y) = \bar{X}\bar{Y}$$

$$f_g(X,Y) = \bar{X}\bar{Y} + \bar{X}Y + XY$$

Module Simplified

- 'easyDecoder_simplified.v'

```
1   module easyDecoder_simplified(a,b,c,d,e,f,g,X,Y);
2
3       input X,Y;
4       output a,b,c,d,e,f,g;
5
6       and(a,~X,~Y);    // a output
7
8       or(b,0,0);       // b output
9
10      or(c,X,0);       // c output
11
12      and(d,~X,~Y);    // d output
13
14      or(e,~X,Y);      // e output
15
16      and(f,~X,~Y);    // f output
17
18      or(g,~X,Y);      // g output
19
20  endmodule
```

$$f_a(X,Y) = \bar{X}\bar{Y}$$

$$f_b(X,Y) = 0$$

$$f_c(X,Y) = X$$

$$f_d(X,Y) = \bar{X}\bar{Y}$$

$$f_e(X,Y) = \bar{X} + Y$$

$$f_f(X,Y) = \bar{X}\bar{Y}$$

$$f_g(X,Y) = \bar{X} + Y$$

Courtesy:Erin DeLong

Testbench for an Easier Example

- 'testbench4easyDecoder.v'

```
1  module testbench4easyDecoder();
2      reg [1:0] switches;
3      wire [6:0] LED;
4
5      easyDecoder d1(LED[6], LED[5],LED[4],LED[3],LED[2],LED[1],LED[0], switches[1], switches[0]);
6  //  easyDecoder_simplified d1(LED[6], LED[5],LED[4],LED[3],LED[2],LED[1],LED[0], switches[1], switches[0]);
7
8      initial
9          begin
10             switches = 00;
11             $display("switches=XY, LEDs=abcdefg\n");
12             #50 $finish;
13         end
14
15     always
16         begin
17             #10 $display("switches=%b, LEDs=%b", switches, LED);
18             switches = switches + 01;
19
20         end
21
22 endmodule
```

Verilog on Windows

- Make sure you boot your lab computers in **Windows**
  - Open Xilinx ISE 13.2 tools
  - You can follow this tutorial by Brian Stevens

  http://www.csee.umbc.edu/~tinoosh/cmpe650/tutorials/verilog-module-tutorial/VerilogModuleTutorial.pdf

Verilog on The GL Account

- Remotely Accessing the GL Account
- Running Verilog for the 1st time

Remotely Accessing the GL Account

- For windows, download PuTTY

  http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe

  - Open the exe (don't need to install). The Host Name is "gl.umbc.edu" and the Connection type is SSH.
  - Enter your UMBC username and password.

- For Mac/Linux users, open terminal and type

```
ssh <your_umbc_username>@gl.umbc.edu
```

  - Enter your UMBC password.

Running Verilog for the 1st time

1. SSH into UMBC's gl server through Putty, or
   log onto a UMBC computer with Linux and open up a command prompt.
2. Type the following commands:

```
emacs .cshrc
```

3. Go to the bottom and find the last line with an "endif" on it. Below this line, type:

```
source /afs/umbc.edu/software/cadence/etc/setup_2008/cshrc.cadence
```

4. Save this file (ctrl-x ctrl-s), then exit emacs (ctrl-x ctrl-c).

Running Verilog for the 1st time

5. Type the following commands, don't forget the 'dot' at the end of each command:

```
mkdir vhdl
cd vhdl
cp /afs/umbc.edu/software/cadence/etc/setup_2008/cds.lib .
cp /afs/umbc.edu/software/cadence/etc/setup_2008/hdl.var .
cp /afs/umbc.edu/software/cadence/etc/setup_2008/.cdsinit .
cp /afs/umbc.edu/software/cadence/etc/setup_2008/.simrc .
source ~/.cshrc
```

6. To run and compile the 'fred.v' and 'testbench4Fred.v':

```
ncverilog testbench4Fred.v fred.v
```