6.205
(aka 6.111)

Introduction

Fall 2022

Course Overview

- **Prerequisites:** 6.191/6.004 (if no && not co-reqing, email me please. I need a record of that)
- **Units:** 1-5-6
- **Lectures:** Tuesday, Thursday 2:30-4:00 pm in 32-141
- **Labs:** No official time. The lab room is the left (southern) portion of building 38's 6th floor.(38-630) There are about fifty dedicated computers for working on labs and assignments there, however, we also strongly encourage you to install the appropriate toolchains on your own machine when possible.
- **Lab Kit:** You will be provided a Nexys A7 or Nexys 4DDR board for this class which you must take care of and return. This will be used in all labs and will likely form the center of your final project.
- **Piazza:** https://piazza.com/mit/fall2022/6205
- **Textbook:** The internet
- **TAs**:
  - **Jay Lang (jaytlang)**
  - **Fischer Moseley (fischerm)**
- **Instructor:**
  - **Joe Steinmeyer (jodalyst)**

Mostly correct course calendar on front page of course site including office hours

Grades

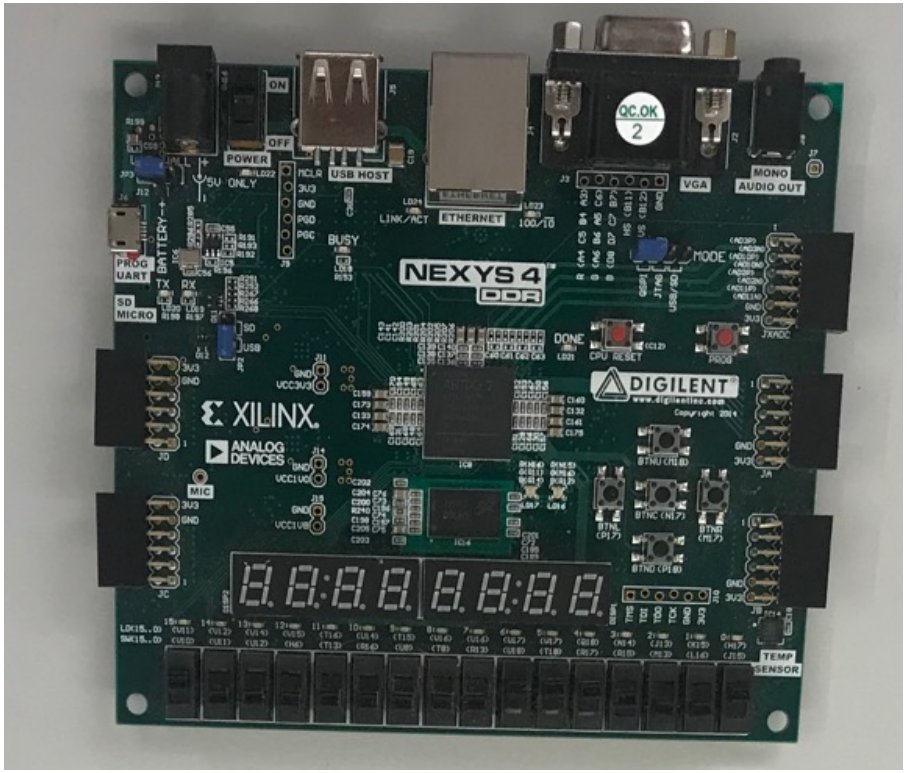
- Your overall grade is based on the following breakdown:
  - Psets: 16%
  - Labs: 34%
  - Participation: 2%
  - Final Project (the project and keeping on track in it): 35%
  - Proposal Presentation and Final Report: 13%
- A large number of students do "A" level work and are, indeed, rewarded with a grade of "A". The corollary to this is that, since average performance levels are so high, punting any part of the subject can lead to a disappointing grade.
- **Labs:** Out/Due Weekly-ish (first one today, due next Thur)
- **Psets:** Out/Due between Lectures (first one today, due Tues)
- **Final Project:** Details in coming weeks

CI-M Possibility

- We do a lot of writing and presenting with our final project.
- I'm trying to see if we can get CI-M status again (not sure no promises for this term)
- Best case we don't have it officially, but you could petition it after the fact.
- Worst case, we don't have it for this term
- No promises! Trying to figure out ASAP

Lab Kit

- Using a Nexys 4 Board
- Additional parts for some labs (pick up as needed)
- Must return at end of semester



Collaboration

- Labs and Psets must be done independently but students may seek help from other students and of course staff.
- Work submitted must be your own
- Violations/copying work will be dealt with seriously. Don’t put us in that position. Nobody (us or you will be happy)

Grade/Lateness Mechanics

- Psets are due when they are due. You can't turn those in late (so don't skip them!)
- Labs are due at 10pm on the due date. Every day late, lose 20 % (doesn't accrue on Sat/Sun)

Office Hours

- Office Hours calendar on the main website page, will be updated weekly by staff
- Office hours in south-side of 38-630, the 6.205 lab
- In person:
  - In-person support will be prioritized by staff
  - Lab machines in 38-630 are open for use by you and have all software installed (~48 machines)
- Checkoffs must be done in person
- Post on Piazza (gets quick responses generally)

6.205's Goal

- We focus on digital design in this class and apply it to FPGAs
- We will want to do lots of simulations, but the end goal is always to have working implementations on hardware!

WTFPGA?

- A giant array of very primitive logic blocks, memory, and other specialized hardware that are each individually programmable.
- The giant array exists in a huge sea of programmable interconnects
- You have full control over all the modules and they can run at the same time; not bound by the fixed structure and limitations of a computer

Where are FPGAs used?

- Anywhere that speed and efficiency are important:
  - Hardware accelerators
  - Stream processors
- Where the "generalness" of a computer isn't needed
- In chip prototyping: After you've simulated a new design but before you spend tens of millions and wait eighteen months to make a chip you will prototype it on an FPGA

Overview Today

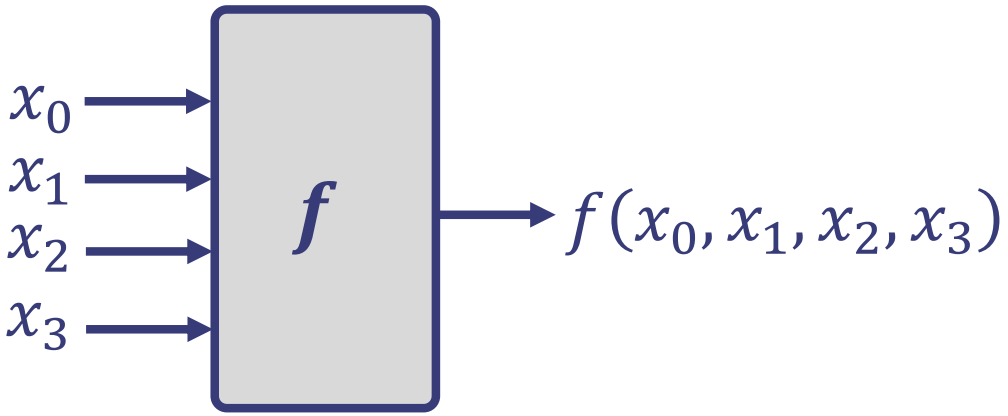
- Logic and Computation
- (System)Verilog:
  - Basics
  - Combinational Logic
  - Modules
- Do a full build cycle with an FPGA (time-depending)

Digital Logic

- Subfield of electronics
- Voltages in the circuit are classified as either “1” or “0”
  - each ”family” of digital logic has its own specifications as to what constitutes a 1 or a 0
- Digital circuits are designed to:
  - interpret input information as 1’s and 0’s
  - generate output information as 1’s and 0’s
- Digital electronics are the reason for the computer revolution:
  - Robust, scalable, inexpensive, etc…

Two Broad Types of Digital Logic

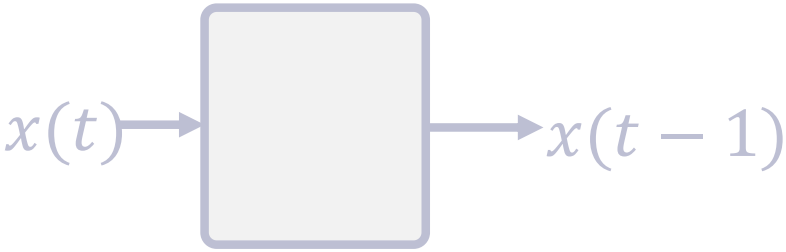
Functions:



Stateless

Current Output is based ONLY on current Inputs NOT a function of time

Storage:



Stateful

Current Output is based past Input

Digital Functions

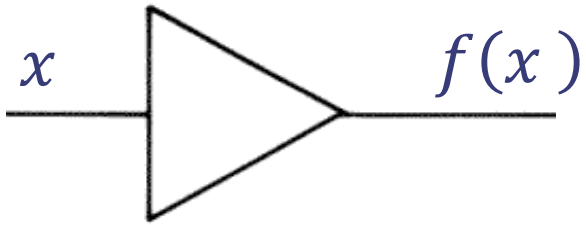
- These work just like $f(x_0, x_1, x_2, x_3)$ from "regular" math, however…
- The values of of inputs and outputs are limited to the Boolean domain $\mathbb{B}$ which is just $\{0,1\}$
  - as opposed to Real Numbers $\mathbb{R}$
  - Or complex Numbers $\mathbb{C}$
- Therefore the "space" of the inputs is often relatively small
- We will often write these using "Truth Tables"
- Commonly used ones functions we will give special names to

1-bit functions (input is a single value):

- How many possible 1-bit functions exist?
- Two (actually 4)…

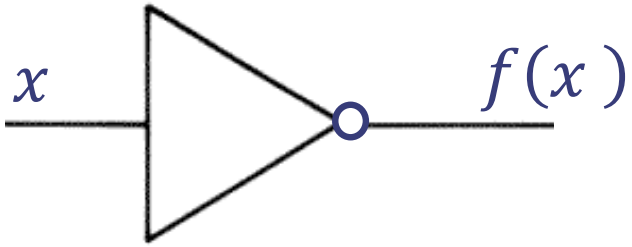
Buffer (Yes) gate:

x	$f(x)$
0	0
1	1



Inverter (Not) gate:

x	$f(x)$
0	1
1	0



Always On gate:

x	$f(x)$
0	1
1	1

Always Off gate:

x	$f(x)$
0	0
1	0

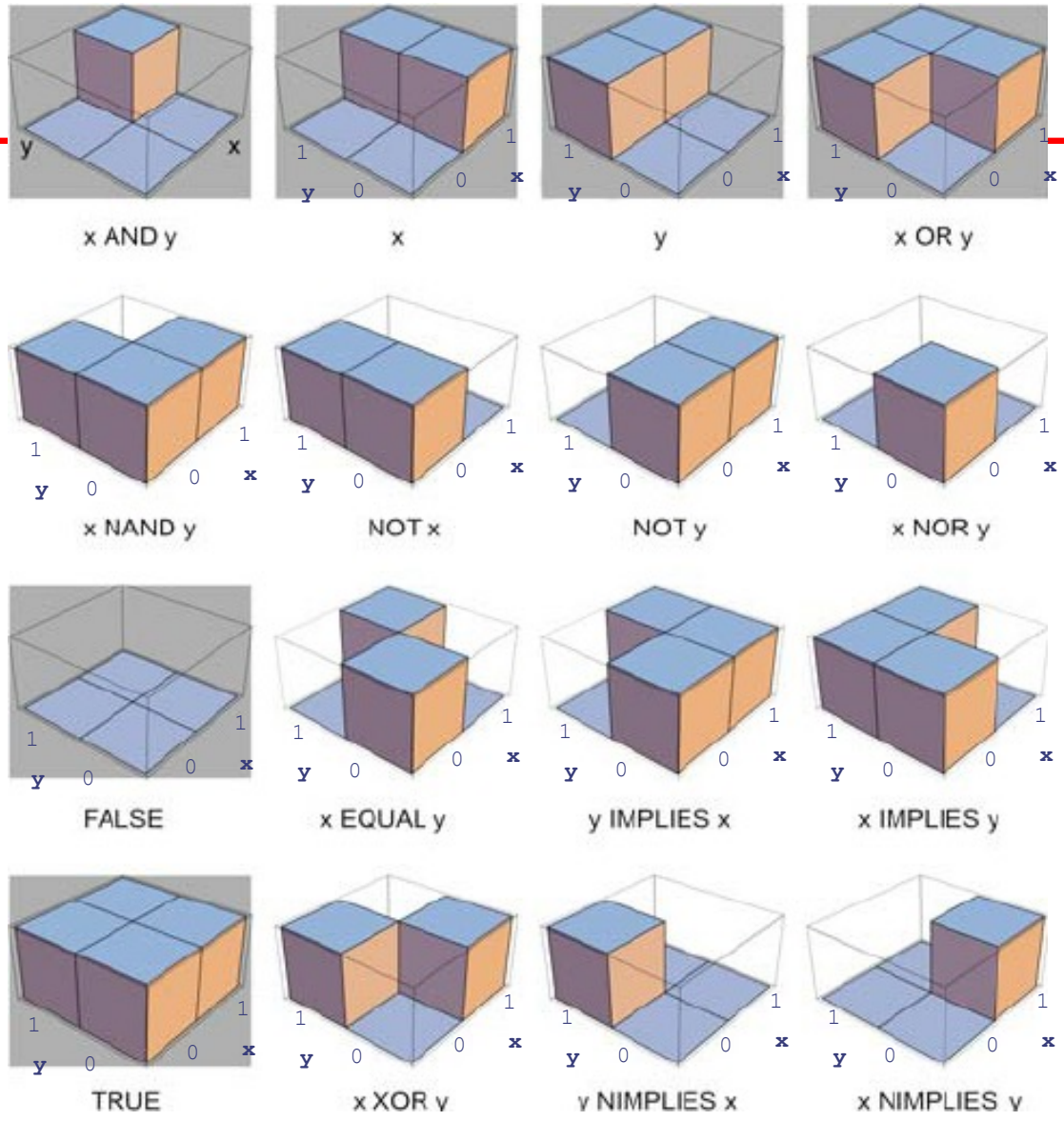
2-bit functions:

$f(x,y)$

x	y	$f(x,y)$
0	0	$f(0,0)$
0	1	$f(0,1)$
1	0	$f(1,0)$
1	1	$f(1,1)$

$2^4 = 16$ possible functions exist

Stated another way: there are 16 unique 1-0 combinations for: $f(0,0)$, $f(0,1)$, $f(1,0)$, and $f(1,1)$



Mayo, Avi & Setty, Yaki & Shavit, Seagull & Zaslaver, Alon & Alon, Uri. (2006). Plasticity of the cis-Regulatory Input Function of a Gene. PLoS biology. 4. e45. 10.1371/journ

Simple Truth Tables

- For a single-input system, there are four possible mappings (two non-negligible)
- For a two input system, you have 4 input combinations and 16 possible truth tables
- There is a lot of complexity that these give us

YES

INPUT A	OUTPUT
0	0
1	1

NOT

INPUT A	OUTPUT
0	1
1	0

AND

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0

NAND

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	1
0	1	1
1	1	0

NOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1

Abels and Khisamutdinov, 2015, https://www.researchgate.net/publication/291418819_Nucleic_Acid_Computing_and_its_Potential_to_Transform_Silicon-Based_Technology

Logical Reduction

- All high level operations we may want can be reduced down to combinations of these simpler logical operations
- We just need to start to see how.
- Don’t just think of the “AND” gate as ”AND”. A lot of things we’d want to do when writing high-level logic/programs rely on it, even if we don’t name it that explicitely. Same with “OR” or “XOR”

Consider just one of these truth tables "XOR"

- If 0 and 1 are numbers, it performs base 2 addition:
  - 0+0=0
  - 0+1=1
  - 1+0=1
  - 1+1=0 (carry 1)

XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

- Or, if 0 means positive and 1 means negative, performs sign determination of multiplication:
  - 0×0 = 0 (positive×positive = positive)
  - 0×1 = 1 (positive×negative = negative)
  - 1×0 = 1 (negative×positive = negative)
  - 1×1 = 0 (negative×negative = positive)

Or still thinking about ways of using XOR

- Or expresses the if/else check:

```
if(A==1):
    output = !B
else:
    output = B
```

- Or it does the check: A!=B
- Or others
- Many high-level algorithmic needs find their basic implementation in these

XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

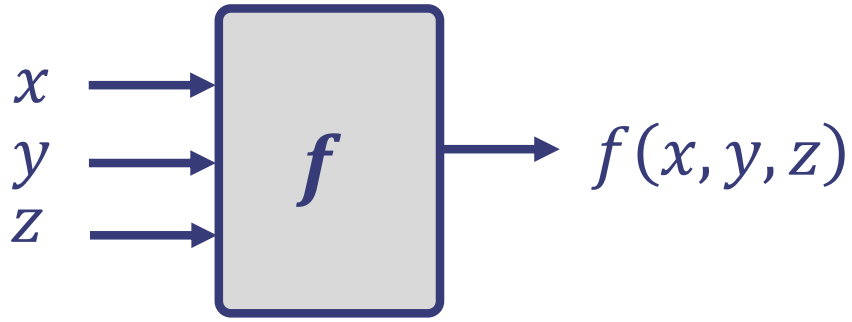
3-bit functions:

$f(x, y, z)$

$2^8 = 256$ possible patterns for f

Space of a function is based off its input width:

$2^{2^n} = 2^{2^3} = 2^8 = 256$



x	y	z	$f(x, y, z)$
0	0	0	$f(0,0,0)$
0	0	1	$f(0,0,1)$
0	1	0	$f(0,1,0)$
0	1	1	$f(0,1,1)$
1	0	0	$f(1,0,0)$
1	0	1	$f(1,0,1)$
1	1	0	$f(1,1,0)$
1	1	1	$f(1,1,1)$

More Complex Logic Functions

- 3 input Truth table:
  - A,B,C can be a three bit number:

```
if {A,B,C}==7:
 Z=1
else:
 Z=0
```

  - A,B two-bit number, C some condition:

```
if {A,B}==3 and C:
 Z=1
else:
 Z=0
```

  - Etc…

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

https://electronicspost.com/explain-logic-and-gate-and-its-operation-with-truth-table/

https://reviseomatic.org/help/e-logic/Logic%20Truth%20Tables.php

Things Scale Quickly

- As you add more and more bits to your function input, the number of possible functions you can express grows astronomically…

Number of n bit functions that exist: 2^{2^n}

6-bit functions: $f(x_5, x_4, x_3, x_2, x_1, x_0)$

64 rows

x_5	x_4	x_3	x_2	x_1	x_0	$f(x_5, x_4, x_3, x_2, x_1, x_0)$
0	0	0	0	0	0	$f(0,0,0,0,0,0)$
.	.	.	.	.	.	…
1	1	1	1	1	1	$f(1,1,1,1,1,1)$

Possible functions you can express with six bit input is:

$2^{2^6} = 2^{64} = 1.84 \times 10^{19}$

Our FPGA has 63,400 6-input logic functions we can independently specify. So our FPGA is quite flexible

This…is Sort of Backwards

- A modern digital engineer usually doesn't start with a set of truth tables and then assign meaning to them. (what we just did)…have a hammer looking for nails.
- We usually want to go in the *other* direction:
  - Describe some sort of logical behavior (if this, then that, etc…)
  - Figure out the most efficient underlying digital function that will express all of that for us and use it!
- We want to synthesize logic we don't want to explain/justify logic

Sum of Products

x	y	z	(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- One way to specify a Boolean function can be in an algebraic expression (| is or, implicit AND):

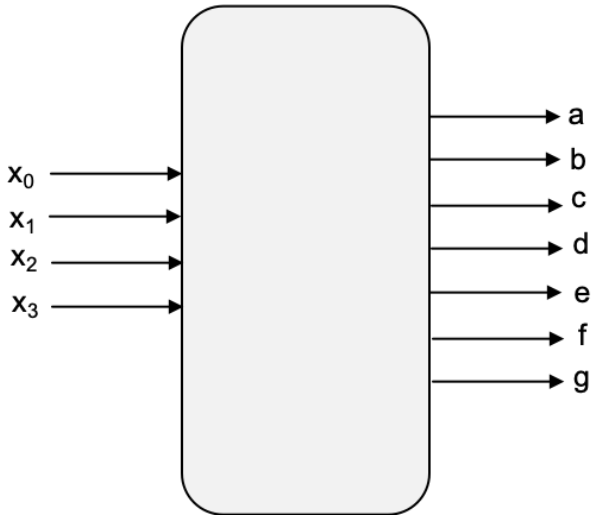
$$f(x,y,z) = \bar{x}\bar{y}\bar{z}\,|\bar{x}\bar{y}z|x\bar{y}\bar{z}|x\bar{y}z|xyz$$

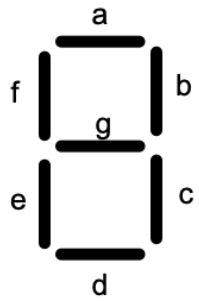
- Function is High when:
  - x is low and y is low and z is low OR
  - x is low and y is low and z is high OR
  - etc…
- Called a "Sum of Products"

Called that because OR is often drawn as + in Boolean algebra, and & is often drawn as multiply

Pset 01

- In PSET 01 you'll design a 7-segment decoder:
  - Four bit digital value in
  - Seven values out which drive 7-segment LED
  - Approach like 7-in parallel sum of products



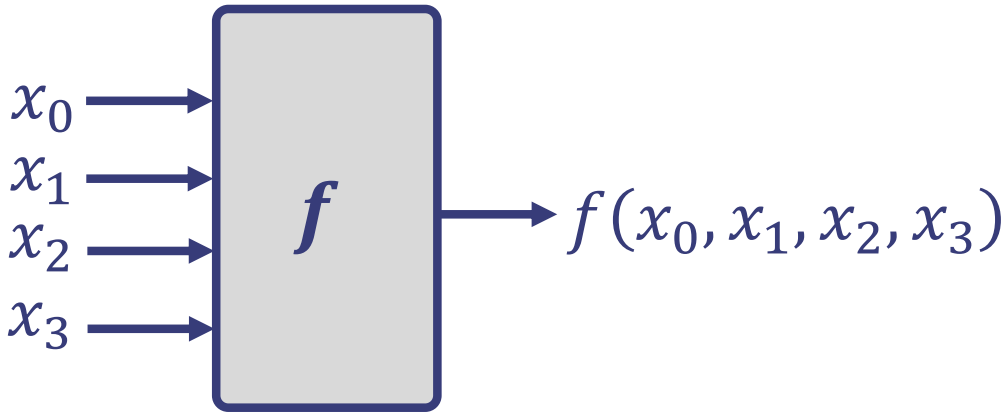


Other Times…

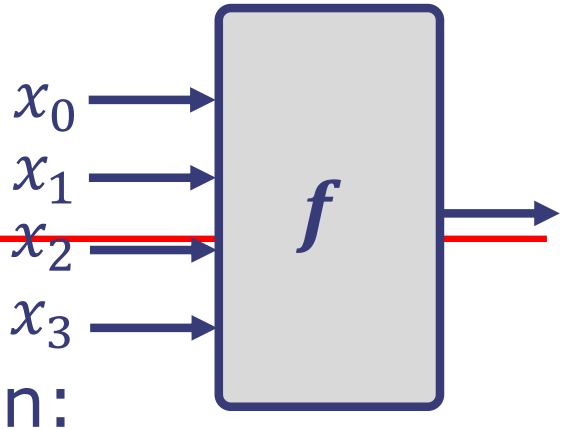
- We want to be able to say: "If x is < 258 make an output go high, but only if button B is not pushed and also do this other thing, but only if Buttons C1 through C18 are in a 0x2AAAA pattern…"
- In that case we want to use higher level constructs and this is really where Hardware Description Languages come in (though you can do sum of products/truth-table stuff in them too).

Logic…(Issues with Time)

- Theoretical Idea of a digital function is divorced from concept of time, but…
- Everything takes time to occur because of capacitance, non-idealities, c:



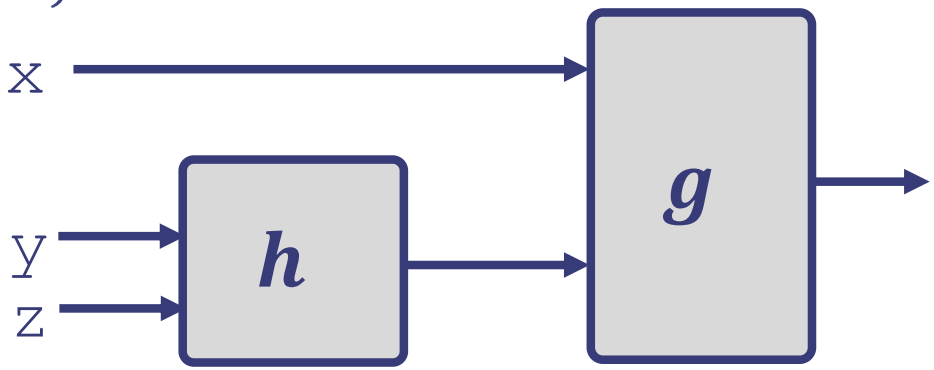
Time???



- Two big numbers for a Digital Function:
  - t_{cd}**: Contamination Delay:** The *minimum* time it takes from an input change on a function to appear at output of function
  - t_{pd}**: Propagation Delay:** The *maximum* time it takes from an input change on a function to appear at output of function
- Therefore, when a new input is presented to a digital (combinational function), it will take between t_{cd} and t_{pd} for it to calculate!
- Can also think of these as best/worst case

Time???

- When functions start to call other functions, the time constraints start to stack:
  - $f(x, y, z) = g\big(x, h(y, z)\big)$



- f's $\boldsymbol{t_{cd}}$ and $\boldsymbol{t_{pd}}$ will be based on combinations of h's and g's $\boldsymbol{t_{pd}}$ and $\boldsymbol{t_{cd}}$:
  - What's the worst worst case?
  - What's the best best case?

SystemVerilog

An Introduction

Verilog: A Hardware Description Language (HDL)

- Verilog is can/be two things:
  - A language used to describe hardware (synthesizable)
  - A language used for simulation of hardware (simulatable)
- The bulk of your work will be in designing Verilog to synthesize onto a device (Actually build something)
- However you will use other Verilog (testbenches) to test the first kind in simulation!
  - The synthesizable type should not be thought of as a "programming language" like Python or C, rather a descriptive language for blueprints about hardware. This Verilog is "interpreted" and makes a schematic.
  - Testbench Verilog *can/should* be though of like Python or C…This code actually "runs" like code.

Variables in Verilog

- In Verilog:
  - `logic` is one type of variable
  - *How* a `logic` gets *used* determines what it represents

```
//variables:
logic a; //create one
logic b; //create another
logic c,d,e; //create several at same time
```

Variables in Verilog

- In Verilog we have flexibility to specify the size of variables:
  - By default things are one bit, but you can specify sizes like shown below (sizing specified left to right):

```
//variables:
logic a; //create one bit variable
logic [3:0] b; //create four bit variable
logic [11:0] c,d,e; //create several 12 bit variables
```

- Why not just use standard types (32 bit int for example)?
  - You can, but unless needed, why waste the resources?
  - If you need four bits, just use four bits

Values in Verilog

- Good practice to always specify values in the following form: **S'Txxxx_xxxx** where
  - **S** is the size of the number (in bits)
  - **'** is the single quote marker
  - **T** is the numerical base you're specifying the value in
    - b for binary (0,1)
    - d for decimal (0,1,2,3,4,5,6,7,8,9)
    - h for hex (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
  - **xxxx_xxxx** are your values
    - The _ is ignored in evaluation
    - use _ to make more readable
    - Don't need to use _ but is really nice

Values in Verilog

- Some examples:

```
10'b01010_01010_00; //10 bit size of value...
10'b1; //10 bit value but only lsb specified...so this is saying 10'b0000_0000_01;
12'hF0F; //12 bits..this would be 12'b1111_0000_1111;
9'hF0F; //9 bits so 9'b1_0000_1111; top three cut off since we said only 9 long
15; //assumed to be an 32 bit integer by default:
    //              'b0000_0000_0000_0000_0000_0000_0000_1111;
```

Values of Bits

- Each bit can take on four values:
  - 1: Logical 1
  - 0: Logical 0
  - X: Undefined
  - Z: High Impedance

Order of Operations in Verilog

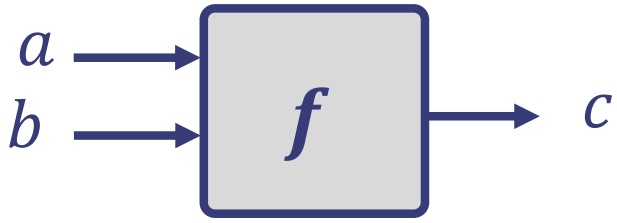
- Be careful!
- The order is not always what you expect!
- Use parentheses for safety!

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality
=== !==	case equality case inequality	equality equality
&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

https://class.ece.uw.edu/cadta/verilog/operators.html

Creating Digital Functions

- Two ways:
  - Using `assign` statements
  - Using `always_comb` blocks
- Let's say I wanted `c = b || a` aka "c is b OR a"




```
//create combinational...
//element with assign:
logic a, b, c;
assign c = a || b;
```

```
//create combinational element...
// with always_comb block:
logic a, b, c;
always_comb begin
  c = a || b;
end
```

Will result in a piece of combinational logic that carries out this function:

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

What about "Higher Level" Constructs?

- If/Else Statements?
- Example: *Have three one-bit inputs, if any two are 1, the output is 1 (aka a "majority function")*

x	*y*	*z*	*output_1*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Majority Function Solution(s)

```
logic x,y,z,output_1;
//one way (“Sum of products”):
assign output_1 = (!x && y && z)|| (x && !y && z) || (x && y && !z) || (x & y && z);
//another way: ('b is base header for binary, so 'b011 means 011 in binary)
assign output_1 = ({x,y,z}=='b011)||({x,y,z}=='b101)||({x,y,z}=='b110)||({x,y,z}=='b111);
//another way (chained ternary operator)
assign output_1 = {x,y,z}>=5?1:{x,y,z}==3?:1:0;
//numbers not specified with base header default to base 10!
//ternary chain above except done with alway_comb block
always_comb begin
  if ({x,y,z}>=5))begin
    output_1 = 1'b1;//specify bit
  end else if ({x,y,z}==3'b101) begin
    output_1 = 1'b1;
  end else begin
    output_1=0;
  end
end
//ternary chain above except done with alway_comb block (different)
always_comb begin
  output_1=0;
  if ({x,y,z}>=5))begin
    output_1 = 1'b1;//specify bit
  end else if ({x,y,z}==3'b101) begin
    output_1 = 1'b1;
  end
end
```

Modules

- Use modules to compartmentalize/reuse your code:

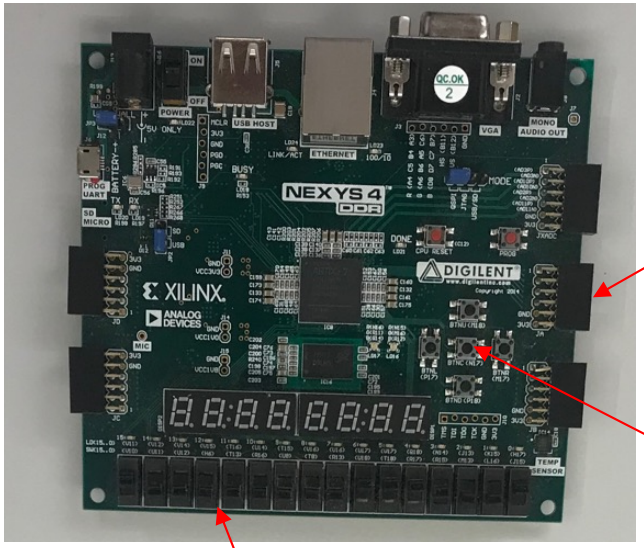
```
module f(input wire x, input wire y,
         input wire z, output logic output_1);
  assign output_1 = ({x,y,z}=='b011)||
                    ({x,y,z}=='b101)||({x,y,z}=='b110)||
                    ({x,y,z}=='b111);
endmodule
```

- Then else where you can make an instance:

```
logic q,r,t,cat;
//declare instance of that module:
f my_f(.x(q), .y(r), .z(t), .output_1(cat));
```

Let's Build Something

- MIT's mascot is the beaver and Unicode symbol for beaver is represented using the hex code **0xF9AB**
- Build a beaver system where if we enter the beaver code correctly and push the beaver button we get a beaver sound



Connect a buzzer to make the beaver sound

Use btnc as beaver button

Use 16 switches to enter beaver code

Step 1

- Make folder/directory to work in:
- Into that folder make the following directories:
  - src: Verilog files that represent actual hardware
  - sim: Verilog files that simulate the src files
  - xdc: File that gives physical FPGA pins ”human” names
  - output_files: where our build results will go

Step 2

- Create a top-level module that will contain our lock module.
- Top-level module has inputs and outputs which are mapped directly to hardware pins

Step 3

- Write the lock module (put in lock.sv)

Step 4

- Write a **testbench** for lock.sv
- This allows us to test lock.sv quickly with little pain

Step 5

- Build!
- Pray to your god!
- Upload!

What's Next

- Get Lab Kit (we'll do after class today or in office hours tomorrow)
- Pset 01 out on site, due by Tuesday.
- Lab 01 out, due next Thursday
  - Lab 01 Requires Pset 01 to be done (use code from pset in lab)