

6.205
(aka 6.111)

Introduction

Fall 2026

Course Overview

- **Prerequisites:** 6.191 (if no, email me please; we will find out)
- **Units:** 1-5-6
- **Lectures:** Tuesday, Thursday 2:30-4:00 pm in 45-230
- **Labs:** No official time. The lab room is the left (southern) portion of building 38's 6th floor.(38-630)
- **Lab Kit:** You will be provided an FPGA board along with a few supplementary items for this class which you must take care of and return. They will be used in all labs and will likely form the center of your final project. You must return them
- **Piazza:** <https://piazza.com/mit/fall2026/6205>
- **TAs:**
 - Heba Hussein (hhussein)
 - Yeabsira Hawaz (yhawaz)
 - M.Subhi Abo Rdan (msubhi_a)
 - Keilee Northcut (keilee)
 - Nerissa Wong (nerwong)
- **Instructor:**
 - Joe Steinmeyer (jodalyst)

Still futzing with course
calendar on front page
of course site including
office hours

Grades

- Your overall grade is based on the following breakdown:
 - Assignments: 42%
 - 7 weeks of exercises and labs before project time
 - Exam: 15% (around week 5)
 - Final Project: 43%
 - Last ~7/8 weeks of semester
- **Final Project:** Details in coming weeks

CI-M

- We are a CI-M
- We do a lot of writing and presenting with our final project. You should be prepared for that.
- Please see syllabus on the course site for details of what is expected.

Lab Kit

- Using a Urbana Board by RealDigital*
- Additional parts for some labs (pick up as needed)
- Must return at end of semester



*newish company

Grade/Lateness Mechanics

- Assignments are come out on Thursdays after class and are due the following Wednesday night.
- Every day late, they 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:
 - The help queue is for in-person work only
 - Lab machines in 38-630 will be open for your use if needed, though everything can be done on laptops.
 - *These machines may not be up with accounts until the weekend or later.*
- **Checkoffs must be done in person, not remotely.**
- Post on Piazza for remote help (gets quick responses generally)

Exam?

- There is an exam this year.
- After many discussions and guidance and trends and reflections, we've decided a nice, air-gapped individual analysis of your understanding of course concepts will be important in this class.
- So there is an exam on ~~October 14th~~ or October 21st at 7:35pm-9:25pm. It will be paper-based, no notes.

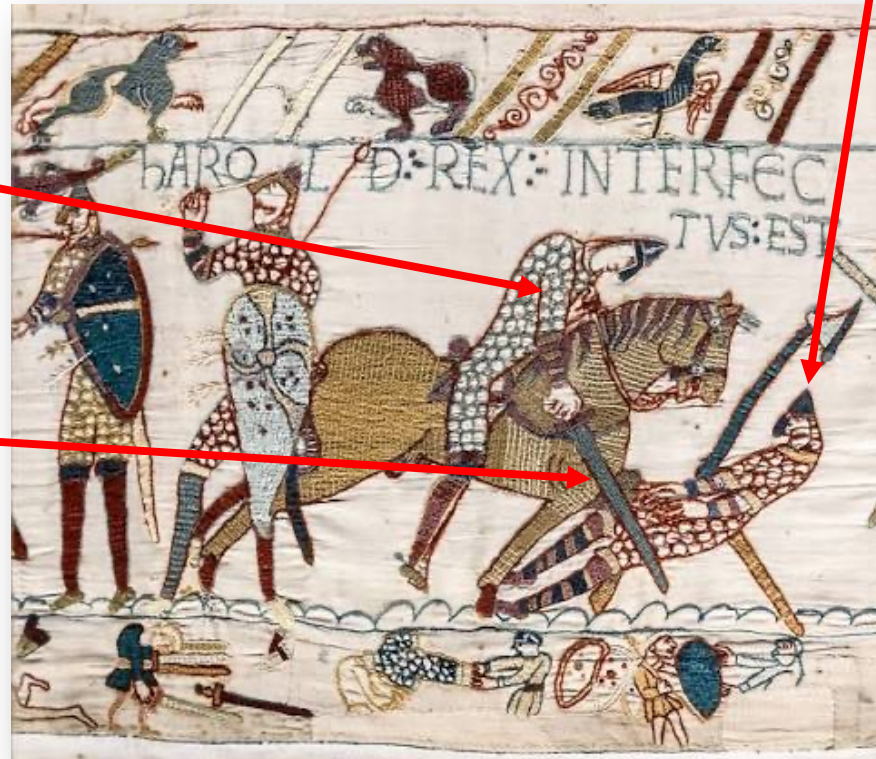
October ~~14~~ 21, 2026

*You
(Student)*

Joe

Exam

- 960 years after the Battle of Hastings



*Prevailing Conditions
wrt education and AI*

Collaboration and Academic Integrity

- Assignments must be done independently but students may seek help from other students and of course staff.
 - That means not copying people's stuff. You can talk with one another, but do not share answers.
- Work submitted must be your own.
- You should **not** be using generative AI to do your assignments. If it is caught, it will be referred to COD and there will be grade impacts including:
 - Getting a score that result in the expression `score==0` evaluating to true
 - `grade <= 'b1111; //F` in hexadecimal

And I get it sorta...

- I am not jealous of your predicament.
- I learned to code and do stuff before it could be handed all to you, so I'm not going to say I understand or experience your temptations.
- But we are in weird times and you only have so many years to set yourself up right.
- The more capable you are of doing things on your own, the more capable you are of doing well in your career.

Hypocrisy...?

- And being in hardware is particularly weird right now....
- I've been in hardware for a while...there were dark dark days for a long time.
- Suddenly our field is hot again and cool in the zeitgeist and gettin' lots of memes.
- And why was that?
- AI

**The more GPUs you buy, the shinier
my jacket gets.**



Hypocrisy...?

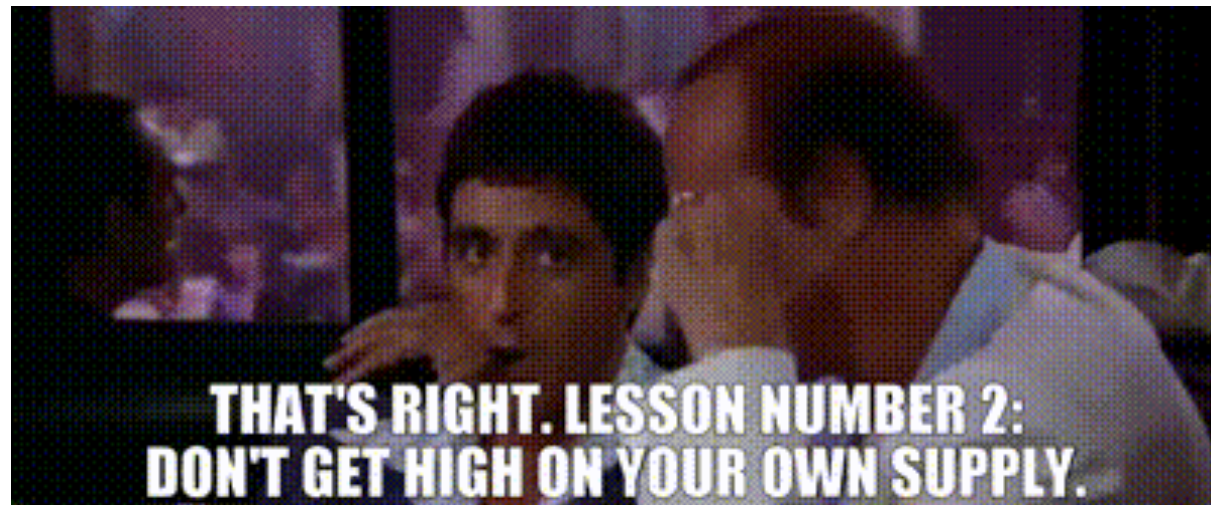
- So who am I to criticize? My/our/your(?) field is having a boom because of the thing I'm cautioning you against.
- What a hypocrite I am!
- But there's a difference between making the thing and subjecting yourself to that thing.

"He knew the poor build the cannons of their own destruction, but it's the rich who sell them."



Or more cynically...

- for the nihilists in the audience...



Do not sabotage yourself

- I think these things coming out are/will be useful if they don't end the world.
- And the people I've seen actually use these things effectively are the people that already know what is going on.
- And I don't think you're there yet.

6.205's Goal

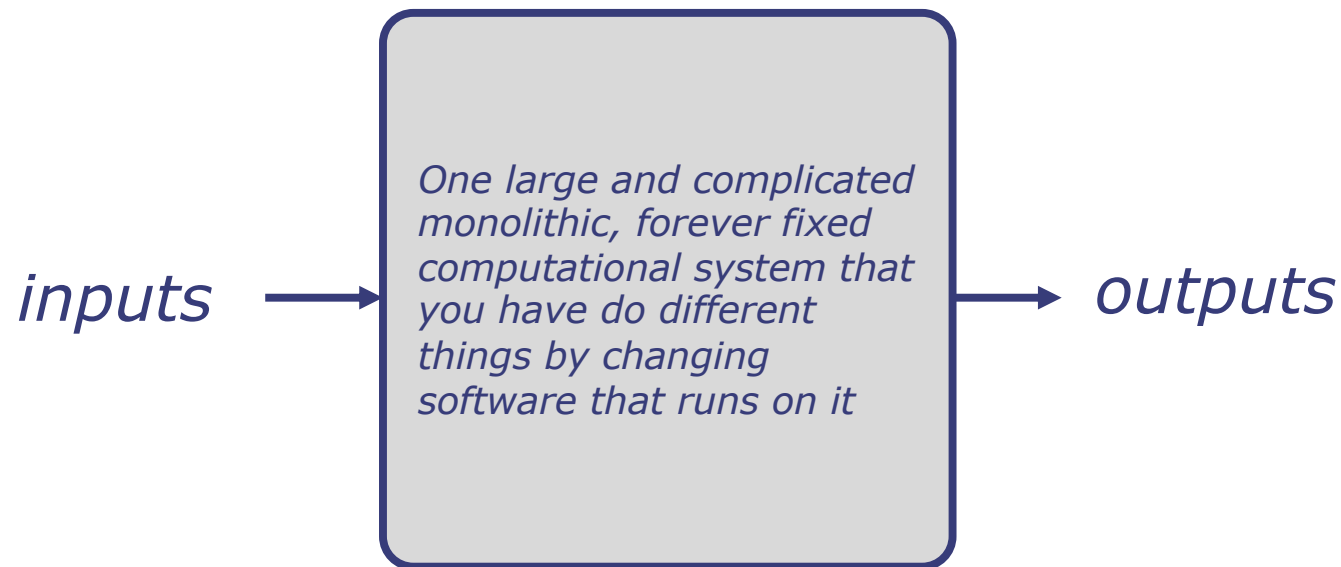
- We focus on digital design in this class and apply it to FPGAs
- Much of a digital designers life is comprised of simulations, but at the end of the day we will want functioning products on our actual device so we strive for working, deployed systems in this class.
- The best bugs and thinkos show up when *actually* trying to get a system working in real life.

WTFPGA?

- A giant **Array** of very primitive logic blocks (aka “**Gates**”, memory, and other specialized hardware that are each individually **Programmable**).
- The giant array of logic exists in a huge sea of **Programmable** interconnects
- The device can be reprogrammed repeatedly even once in a device, hence it can be programmed “in the **Field**”.

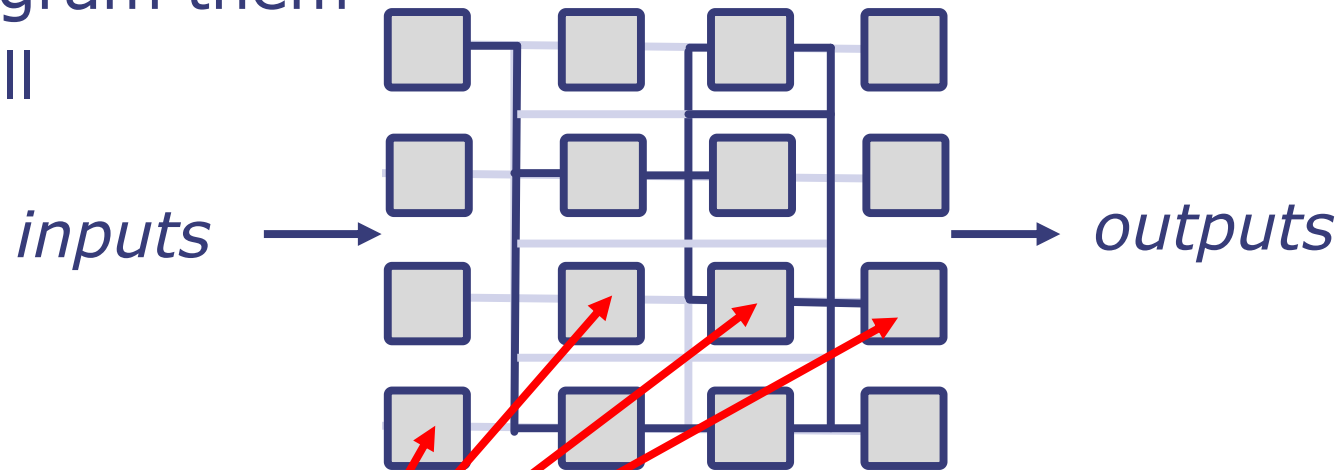
A Regular Computer

- Write Code
- You Program a piece of hardware with that code
- That hardware runs the code



An FPGA

- Many small programmable blocks
- Write code for each and program them
- Write “code” that specifies how they work together and program them
- Run it all



Many small, relatively simple computational elements you individually program and individually connect together as you see fit!

Our cheapo* FPGA in 6.205

- Has roughly 8,500 programmable slices each with four programmable logic blocks
- 150 18kb memory units
- 5 clock management tiles
- 120 dedicated multipliers
- Vast array of input, output devices
- ...and all of these can be interconnected as we see fit to build complex systems.

*even cheap FPGAs have resources orders of magnitude more than computation systems from previous decades

Where are FPGAs used?

- Anywhere that speed/latency and efficiency are important:
 - Hardware accelerators
 - Stream processors
 - Communications systems
- Where the “generalness” of a computer isn’t needed or is harmful:
 - Improvements in:
 - Speed
 - Cost
 - Power consumption
- When lots of simpler tasks to be done in parallel
- In 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

Where are FPGAs **NOT** used?

- Not used in a lot of ML applications (exceptions where very-low latency inference is needed). GPUs largely dominate that space because of economies and software infrastructure (e.g. CUDA)
 - Exception is they are used to support a lot of ML/AI stuff (moving data, etc...)
- Anywhere where a dedicated processor *is* advantageous...like in a general computer...or a dedicated GPU is advantageous
- Where economies of scale allow dedicated chips (ASICs) to be financially viable*

The State of Hardware

- Since the death of Moore's Law, we can't just wait for continual gains from shrinking the same old designs.
- There has been renewed push to:
 - Find new computational architectures
 - Build new custom chips in areas neglected
 - Have much more of a hybrid computational environment
- Hardware and digital design is in a very exciting position right now.
- FPGAs can be a good entry point into the field.

Digital Logic

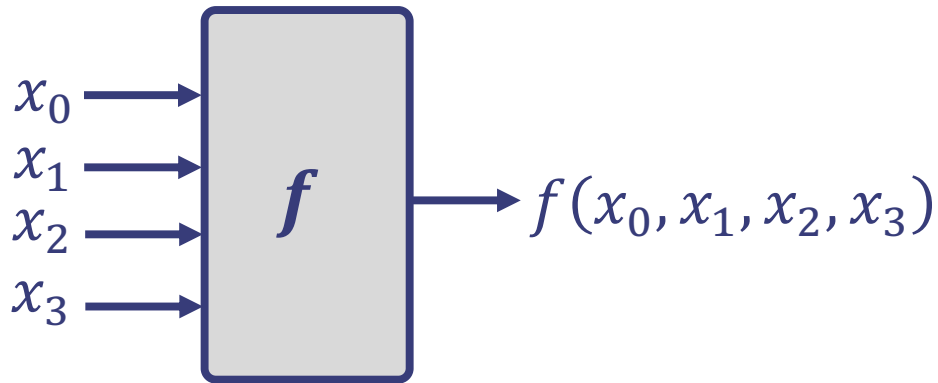
- Subfield of electronics
- Voltages in the circuit are classified as either “1” or “0”
 - each “family” of digital logic has its own specifications as to what constitutes a 1 or a 0
- Digital circuits are designed to:
 - interpret input information as 1’s and 0’s
 - generate output information as 1’s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

Why Does Computation Take Time?

- There is some innate causality at play and
- Down at the core, you have the “Quantum Speed Limit”

Two Broad Types of Digital Logic

Functions:



Stateless

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

Storage:



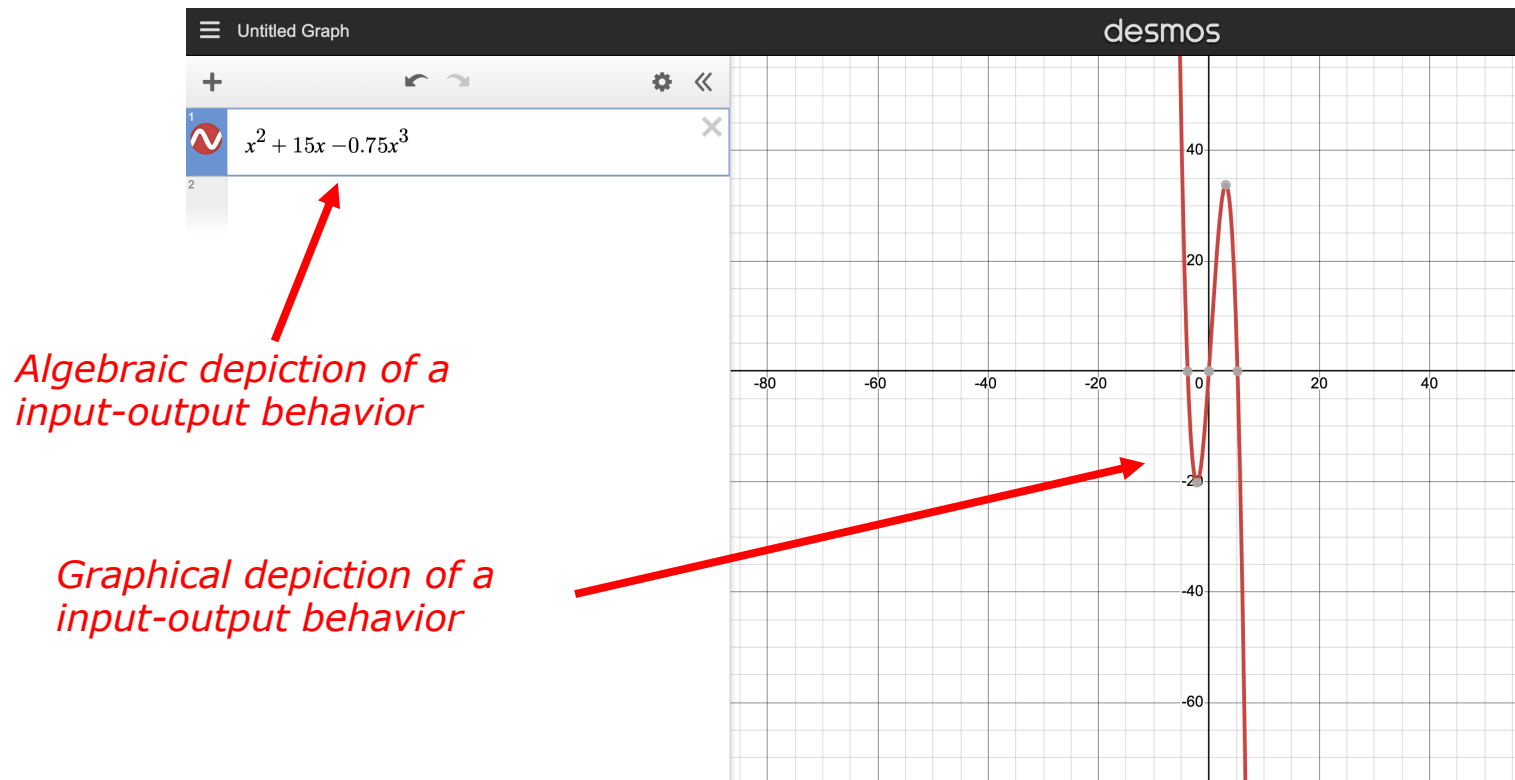
Stateful

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 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 $\mathbb{C}$
- How do we “display” a function that exists in $\mathbb{R}$?
- For example, if I wanted to show you some $f(x)$ in $\mathbb{R}$? How would we do it?

We Plot it or Write it Out with a set of math algebra



Digital Functions

- Digital Functions can also be written using algebraic expressions, just ones limited to the Boolean realm.
- Digital functions can also be graphically depicted, just usually not with graphs...instead we'll use tables.

Boolean algebra

- variables can only be 0 or 1*
- Commonly used operations aren't the same as in the real domain:
 - $\bar{x}$ means "Not x " or the opposite
 - $|$ means logical "or" (sometimes written as $+$)
 - $\&$ or adjacency means "and" (sometimes written as $\cdot$)
- So the algebraic expression of a certain digital function could be:

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

*exceptions that we'll learn about

Digital Functions

- The “space” of the inputs is often relatively small compared to other domains and is therefore not best conveyed using plots
- Instead you will often write these using “Truth Tables”

Express your function algebraically like this:

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

Or “graphically” like this:

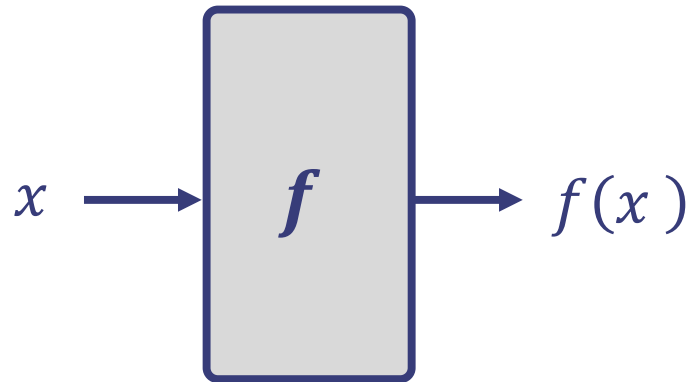
x	y	z	$f(x, y, z)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Digital Functions

- As we'll see, for a low-count variable inputs, there are infinite possible functions in the Real domain $\mathbb{R}$ but only a handful of possible in the Boolean domain $\mathbb{B}$
- For small input functions, the possibilities are so few we give them names even in the digital space.

The Simplest Digital Function Class

- One Bit Input:



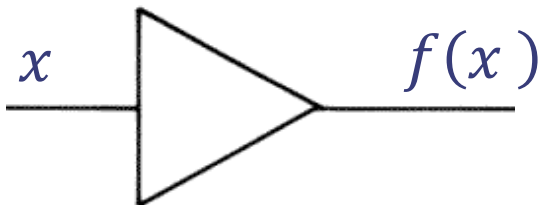
- How many possible 1-bit functions exist?

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

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

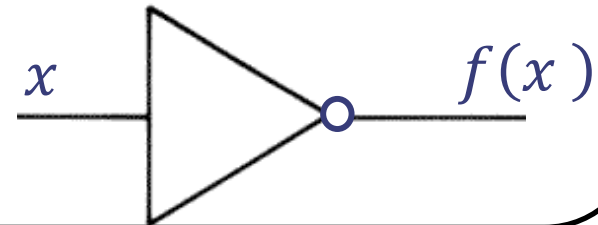
Buffer (Yes) gate:

x	$f(x)$
0	0
1	1



Inverter (Not) gate:

x	$f(x)$
0	1
1	0



Always On gate:

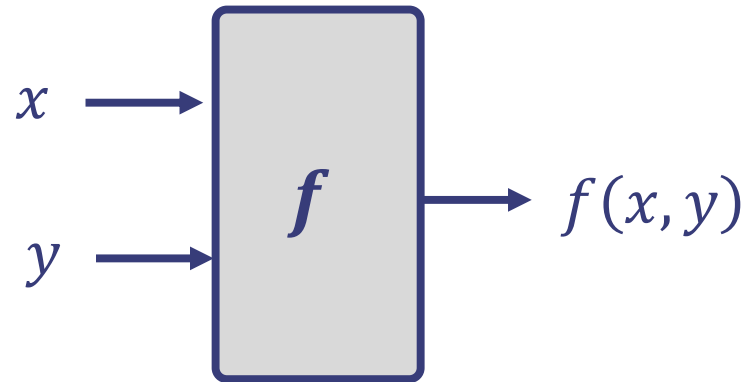
x	$f(x)$
0	1
1	1

Always Off gate:

x	$f(x)$
0	0
1	0

What About Two bits input?

- Two Bit Input:



- How many possible 2-bit functions exist?

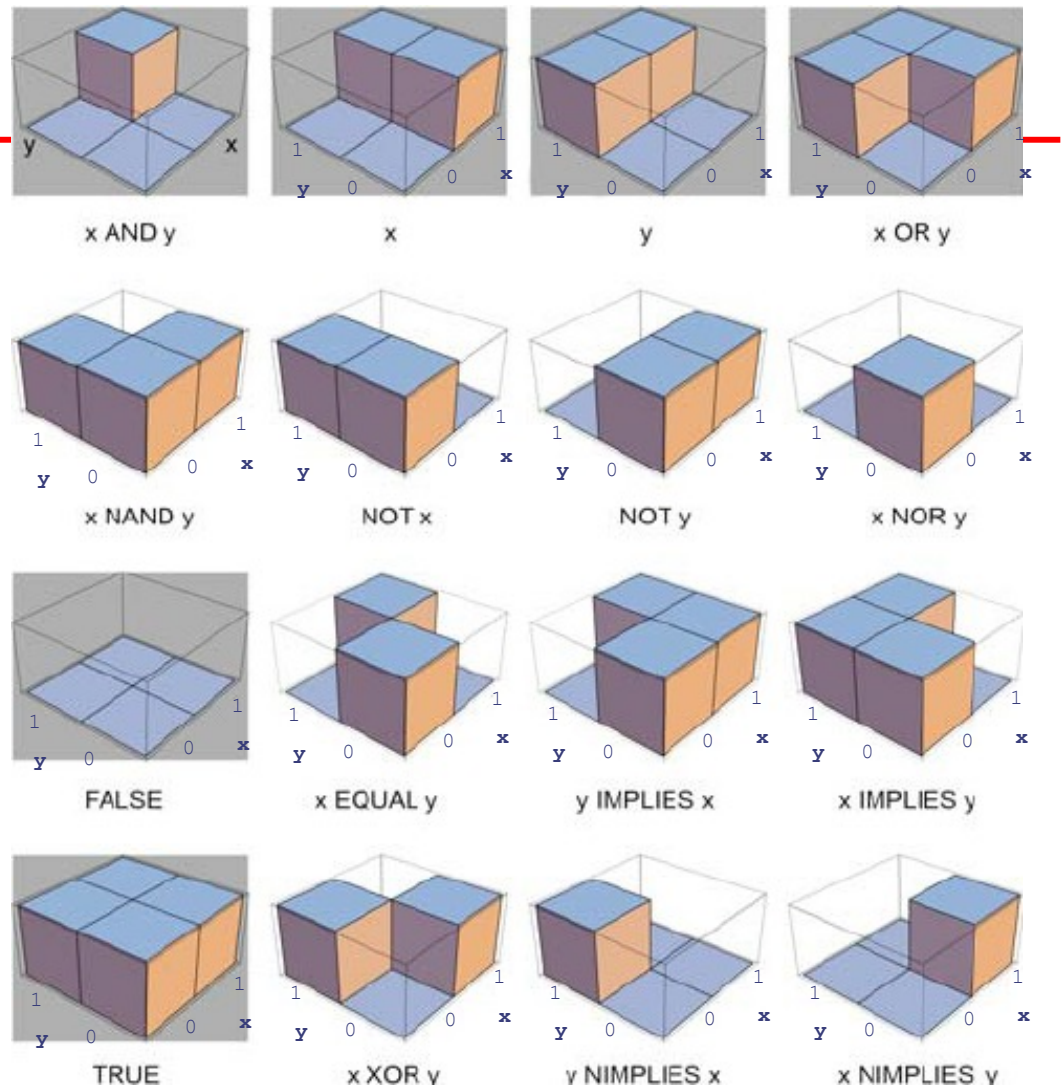
2-bit functions:

$$f(x, y)$$

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/journal.pbio.0040045.

September 10, 2026

<https://fpga.mit.edu/6205/F26>

L01-36

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



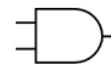
INPUT		OUTPUT
A		
0		0
1		1

NOT



INPUT		OUTPUT
A		
0		1
1		0

AND



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

OR



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

XOR



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

NAND



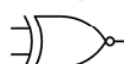
INPUT			OUTPUT
A	B		
0	0		1
1	0		1
0	1		1
1	1		0

NOR



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

XNOR



INPUT			OUTPUT
A	B		
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](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" in the quasi-grammar sense of the term. 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 explicitly.
- Same with "OR" or "XOR"

Consider just one of these truth tables "XOR"

- If 0 and 1 are numbers, XOR 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, XOR performs sign determination of multiplication:
 - $0 \times 0 = 0$ (positive $\times$ positive = positive)
 - $0 \times 1 = 1$ (positive $\times$ negative = negative)
 - $1 \times 0 = 1$ (negative $\times$ positive = negative)
 - $1 \times 1 = 0$ (negative $\times$ negative = positive)

Still thinking about ways of using XOR

- XOR expresses the if/else check:

```
if(A==1):
```

```
    output = !B
```

```
else:
```

```
    output = B
```

- XOR it does the check: $A \neq B$
- XOR does many others
- All high-level algorithmic needs find their basic implementation in these fundamental functions of AND, XOR, NAND, etc...

XOR



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

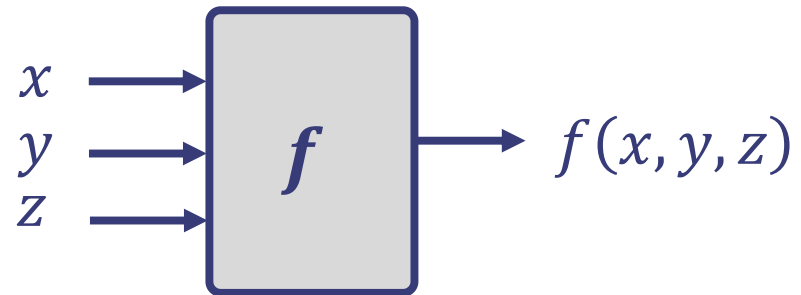
3-bit functions:

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

$2^8 = 256$ 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	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

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

- if {A,B}==3 and C:

- Z=1

- else:

- Z=0

- Etc...

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

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

<https://fpga.mit.edu/6205/F26>

Things Scale Quickly

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

Number of n input 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 33,000 individually programmable 6-input logic functions, meaning we have astronomically large possibilities to build.

Now...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)...like having a hammer looking for nails.
- We usually want to go in the **other direction**:
 - Describe some sort of logical behavior that we want (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 don't want to justify logic that we already have
- We want to synthesize logic to suit a purpose,

Creating Truth Tables Manually

$f(x_5, x_4, x_3, x_2, x_1, x_0)$

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	$f(1,1,1,1,1,1)$

- If you can boil your function down to a truth table, it can be used to “program” the small logic functions in an FPGA!
- For simple things this isn’t too hard

Sum of Products

- One way to specify a Boolean function can be in an algebraic expression

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

- 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"

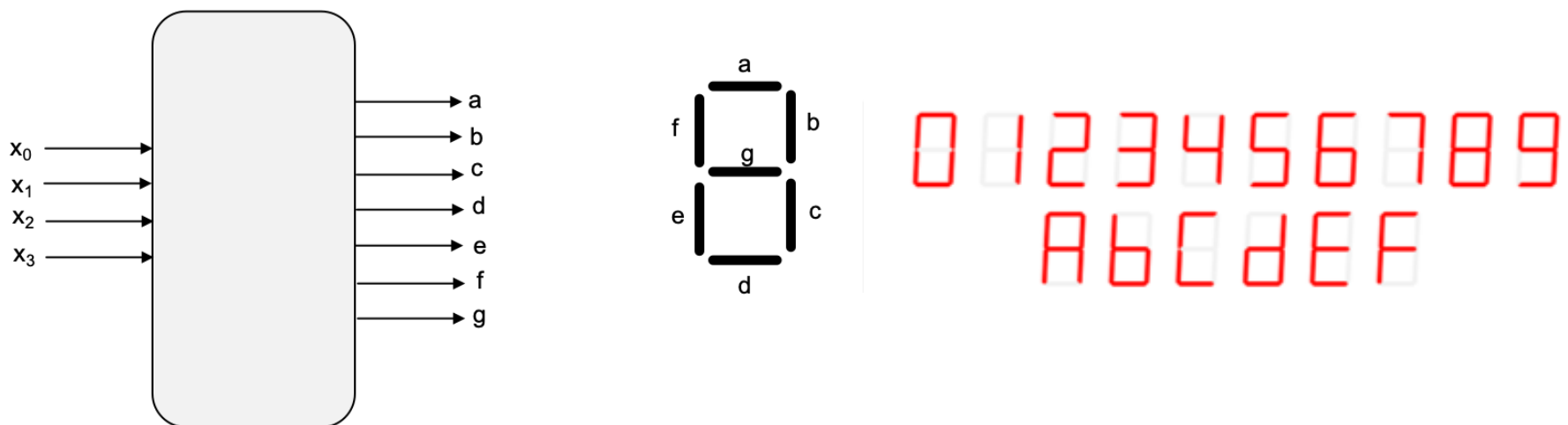
Called that because OR is sometimes drawn as + in Boolean algebra, and & is often drawn as multiply

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

Also sometimes called the
"Canonical Disjunctive
Normal Form"

One of Week 01's Assignments

- In Week 01 you'll design a 7-segment decoder:
 - Four bit digital value in encoding numbers 0 to 15 in binary
 - Seven values out which drive 7-segment LED to
 - Approach like seven separate sum of products.
 - Specify the sum-of-product for the "a" segment...then the "b" segment, etc...

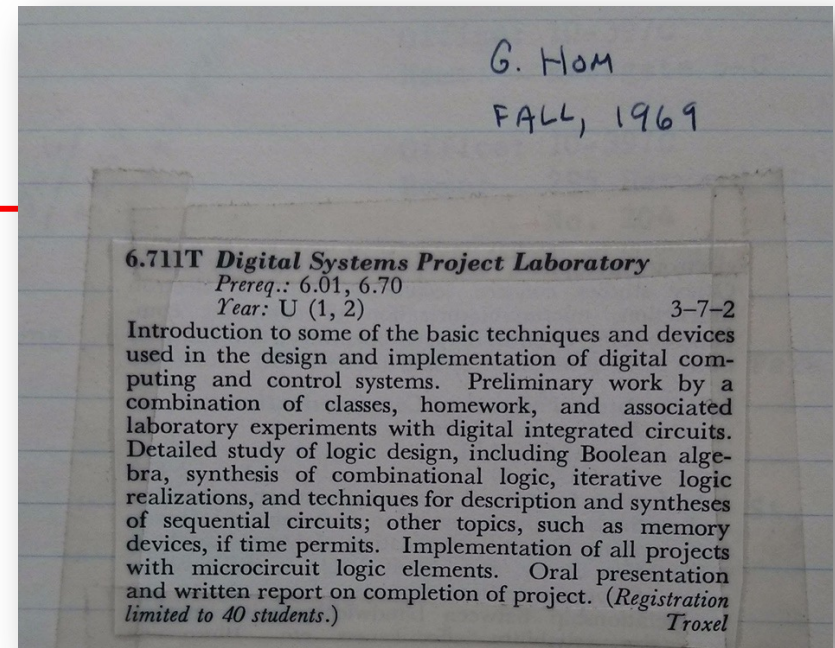


Sum of Products Does Not Scale

- Sometimes, sum of products are the easiest way to just express a digital behavior.
- But 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...”
- You can reduce this logic down to algebraic expressions and truth tables, but *it takes work*

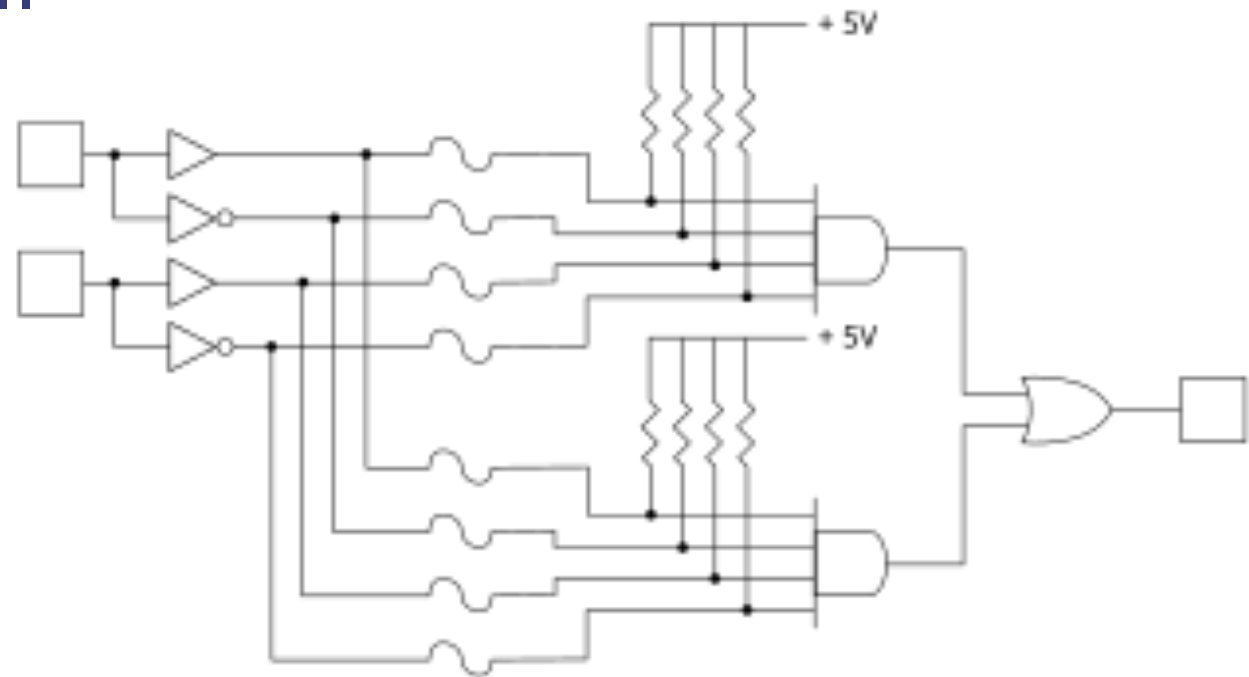
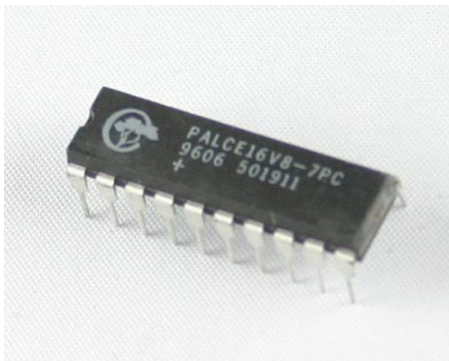
Historically...

- That work of doing all the algebra and optimizing your circuit is what you would do in digital design including in this class.
- 6.711 → 6.111 → 6.205



And Early Proto-FPGAs Implemented Raw SOPs

- Settable fuses that would invert or not invert input signals to feed into AND gates that then feed into OR gates making a sum of product in the process...



Simplified programmable logic device

Sum of Products Does Not Scale

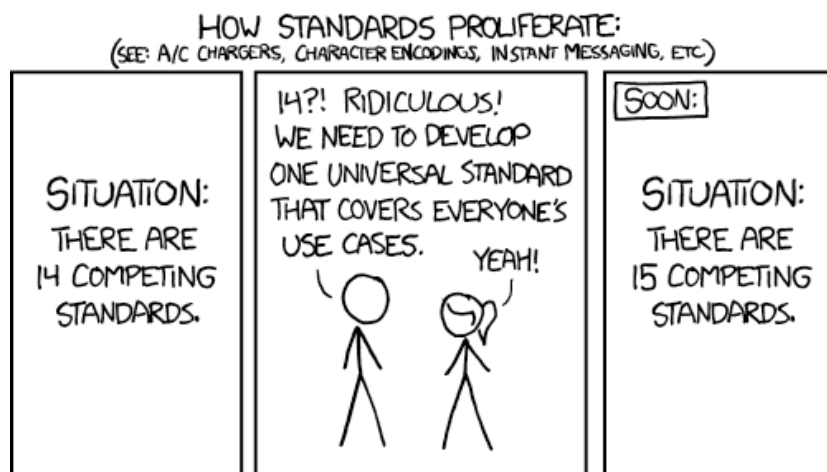
- Sometimes, sum of products are the easiest way to just express a digital behavior.
- But other times...we want to be able to say:
 - “If x is < 258 make an output go high, but only if button B is not pushed and also do this other thing, but only if Buttons C1 through C18 are in a 0x2AAAA pattern...”
- You can reduce this logic down to algebraic expressions and truth tables, but *it takes work*
- Now we use higher level constructs and this is really where **Hardware Description Languages** come in.

SystemVerilog

A Hardware Description Language interprets
high-level algorithmic expressions into low-
level digital logic

Verilog...VHDL...ABEL...Chisel...SystemVerilog...
SystemC...Bluespec...Minispec...Verik...Veryl...
UVM...Amaranth...Lucid...forever and ever
amen.

- We use Verilog and SystemVerilog since it really is an industry standard.
- Yeah... Verilog sucks in lots of ways, but I think it is a good language to know and have some fluency in
- Lots of alternatives, but I'll leave that to you to discover in other situations/classes



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 in 6.205 will be in designing Verilog to ***synthesize*** onto a device
- We'll then use Python for simulating (using a library called cocotb) so we won't use Verilog to simulate much.

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:

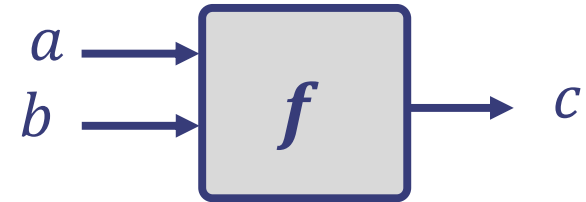
```
10'b1010_1010_00; //10 bit size of value...
10'b1; //10 bit value but only lsb specified...so this is saying 10'b0000_0000_01;
12'hF0F; //12 bits..this would be 12'b1111_0000_1111;
9'hF0F; //9 bits so 9'b1_0000_1111; top three cut off since we said only 9 long
15; //assumed to be an 32 bit integer by default:
    //          'b0000_0000_0000_0000_0000_0000_1111;
```

Values of Bits

- Each bit can take on four values:
 - 1: Logical 1
 - 0: Logical 0
 - X: Undefined (very important)!!!
 - Z: High Impedance

Creating *Pure* Digital Functions

- Two ways:
 - Using assign statements
 - Using always_comb blocks
- Let's say I wanted $c = b \parallel a$ aka “ $c = b$ 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 Boolean 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?
- For Example:
 - *Have three one-bit inputs (x, y, z) and a one-bit output.*
 - *If any two or more inputs are 1, the output is 1*
 - *(aka a "majority function")*

Majority Function Solution #1

```
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);
```

- And Verilog is like C in terms of formatting...use indents to improve readability!!!

```
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);
```

Majority Function Solution #2

```
logic x,y,z,output_1;  
  
//another way (alternative sum of products approach):  
//('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);
```

So readable!

Majority Function Solution #3

```
logic x,y,z,output_1;  
  
//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!
```

*OK...I mean this is maybe a bit better...
This is code where comments would help everyone involved*

Majority Function Solution #4

```
logic x,y,z,output_1;

// same logic as previous ternary chain
// done with always_comb block
always_comb begin
    if ({x,y,z}>=5)begin
        output_1 = 1'b1;//specify bit
    end else if ({x,y,z}==3'b011) begin
        output_1 = 1'b1;
    end else begin
        output_1=0;
    end
end
```

Majority Function Solution #5

```
logic x,y,z,output_1;

//ternary chain previous except done with always_comb block (different)
// this way default sets output_1 to 0 and changes to only on certain conditions
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'b011) begin
        output_1 = 1'b1;
    end
end
// Be very careful with this right now...the way this works can easily lead to confusion
// about things and how to understand System/Verilog
```

Majority Function Solution(s) all together for review

```
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 always_comb block
always_comb begin
    if ({x,y,z}>=5)begin
        output_1 = 1'b1;//specify bit
    end else if ({x,y,z}==3'b011) begin
        output_1 = 1'b1;
    end else begin
        output_1=0;
    end
end
//ternary chain above except done with always_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'b011) begin
        output_1 = 1'b1;
    end
end
end
```

The result of EACH of these lines?

- The logic expressed by this truth table:

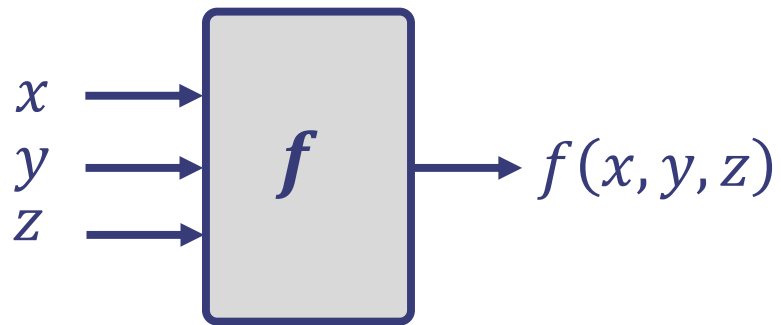
<i>x</i>	<i>y</i>	<i>z</i>	<i>output_1</i>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

* Which can be built using $\frac{1}{4}$ of 1/33000th of our FPGA

- There's often more than one way to write a thing in Verilog

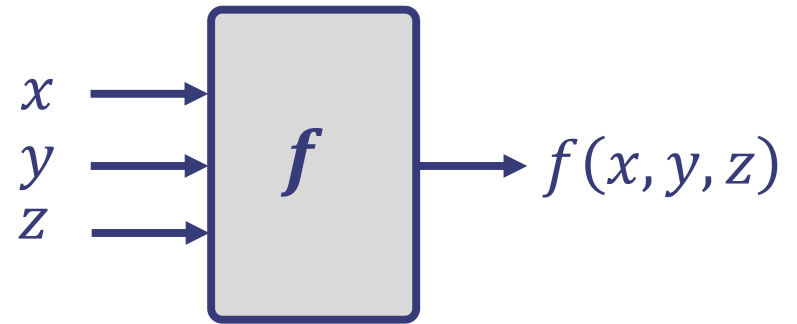
Now...life hits us

- Theoretical Idea of a digital function is divorced from concept of time, but...
- Everything takes time to occur because of capacitance, non-idealities, the speed of light, etc...:



- BIG QUESTION. If you set x, y, z , how long until you see the correct result at the output?

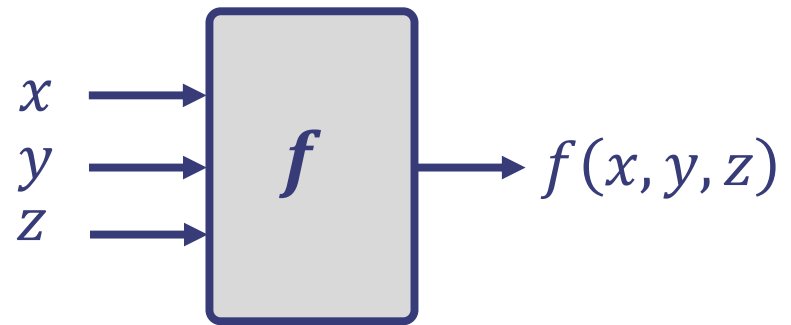
Delays



- Two big numbers for a Digital Function:
 - t_{cd} : **Contamination Delay**: The minimum time it takes from an input change on a function to affect the output of function
 - t_{pd} : **Propagation Delay**: The maximum time it takes from an input change on a function to affect the 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 response times
- Until t_{pd} has passed, ***the input must be held stable***

How Many Calculations per second?

- We'll often want to feed in one set of inputs and then another and then another and then another, harvesting the outputs as they appear.
- If a module has:
 - $t_{cd} = 1\text{ns}$
 - $t_{pd} = 5\text{ns}$
- How many calculations can we get per second?



Modules

- Use modules to compartmentalize/reuse your code:

```
module f1( 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
```

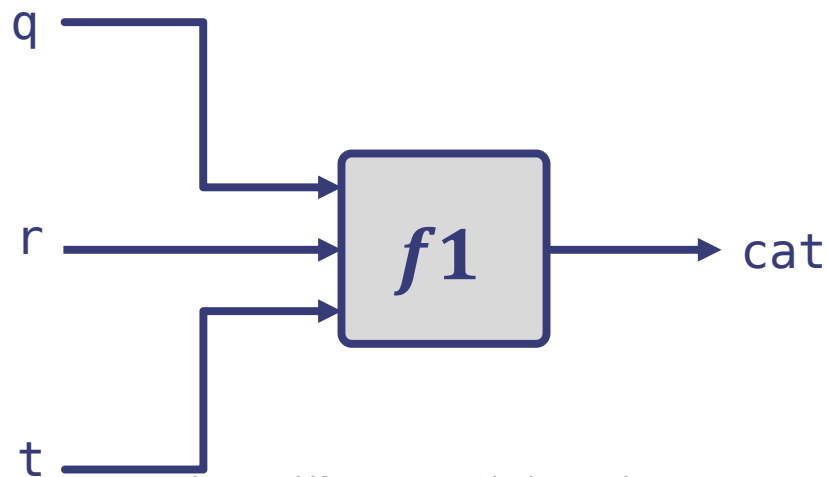
- Then you can make an instance!

Modules

- Use modules to compartmentalize/reuse your code.
- Then you can make an instance:

```
logic q,r,t,cat;  
//declare instance of that module:  
f1 my_f1(.x(q), .y(r), .z(t), .output_1(cat));
```

- Which will build a circuit like this:



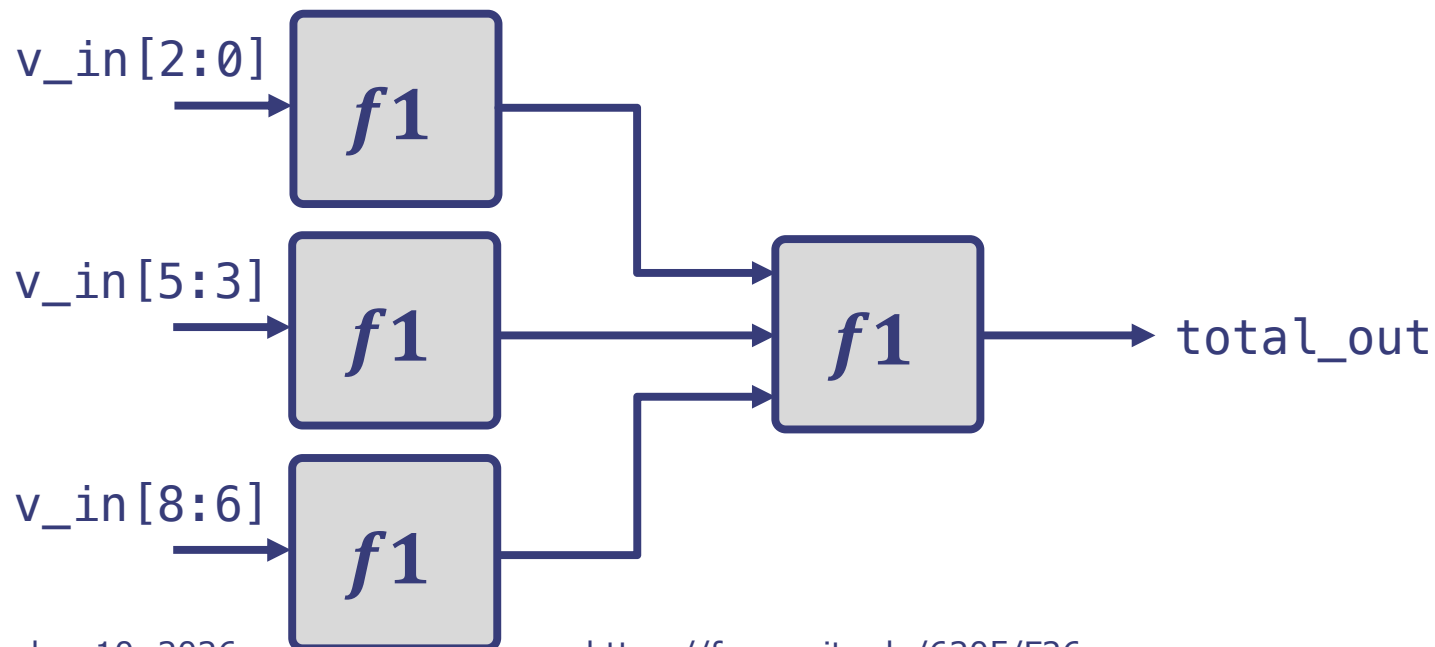
And you can build from here

- What if you wanted a majority function of majority functions?

```
logic [8:0] v_in;
logic [2:0] intermediates;
logic total_out;
//declare separate instances of that same module:
f1 f11(.x(v_in[2]), .y(v_in[1]), .z(v_in[0]), .output_1(intermediates[0]));
f1 f12(.x(v_in[5]), .y(v_in[4]), .z(v_in[3]), .output_1(intermediates[1]));
f1 f13(.x(v_in[8]), .y(v_in[7]), .z(v_in[6]), .output_1(intermediates[3]));
//final layer:
f1 f1total( .x(intermediates[2]), .y(intermediates[1]),
            .z(intermediates[0]), .output_1(total_out));
```

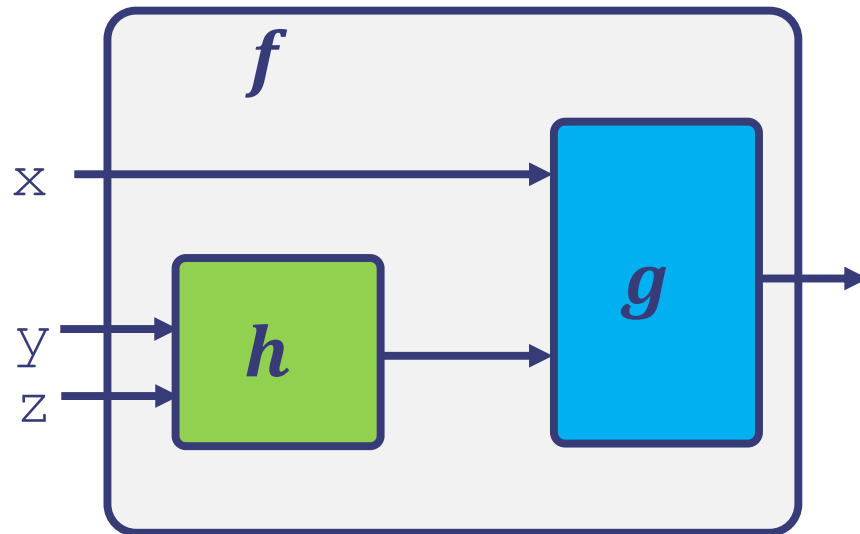
Majority of Majority

```
logic [8:0] v_in;  
logic [2:0] intermediates;  
logic total_out;  
//declare instance of that module:  
f1 f11(.x(v_in[2]), .y(v_in[1]), .z(v_in[0]), .output_1(intermediates[0]));  
f1 f12(.x(v_in[5]), .y(v_in[4]), .z(v_in[3]), .output_1(intermediates[1]));  
f1 f13(.x(v_in[8]), .y(v_in[7]), .z(v_in[6]), .output_1(intermediates[3]));  
  
f1 f1total( .x(intermediates[2]), .y(intermediates[1]),  
           .z(intermediates[0]), .output_1(total_out));
```



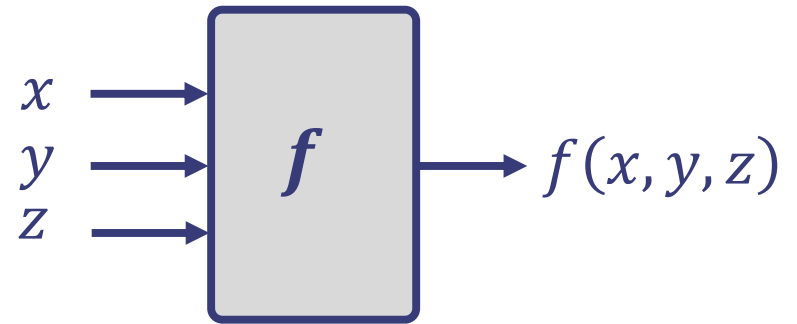
As we start to Attach things...

- Functions Driving Functions... $f(x, y, z) = g(x, h(y, z))$



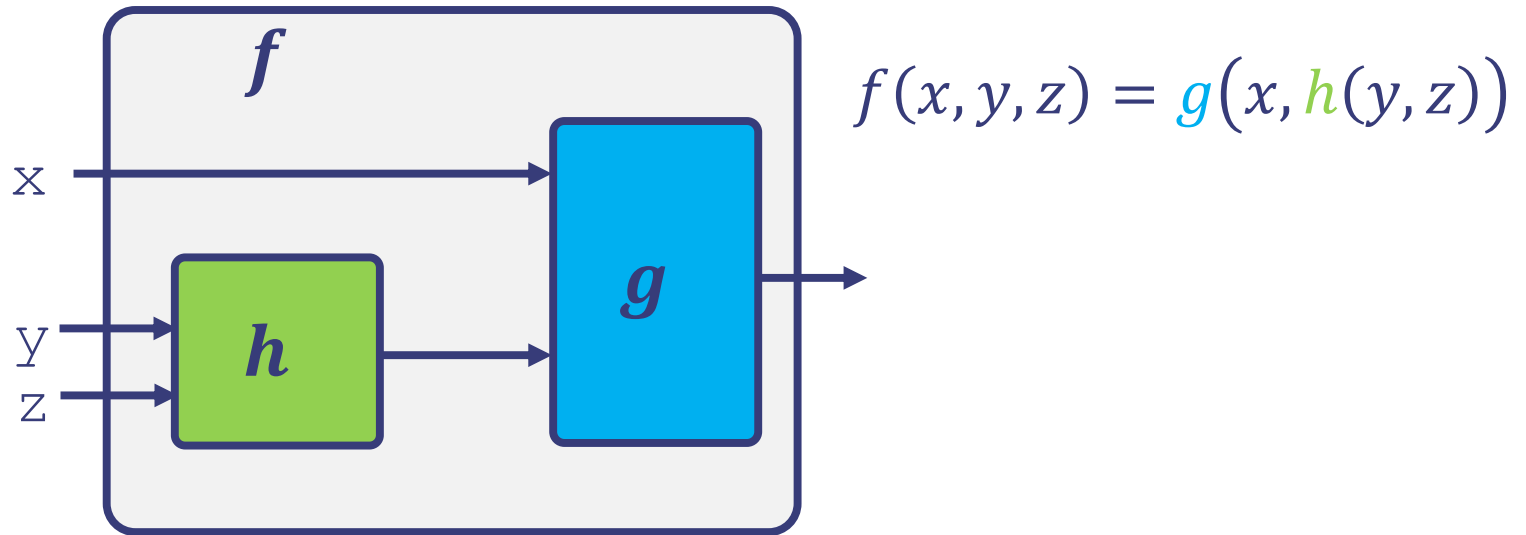
- Any potential problems?

Delays



- 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 response times
- During t_{pd} **the input must be held stable**

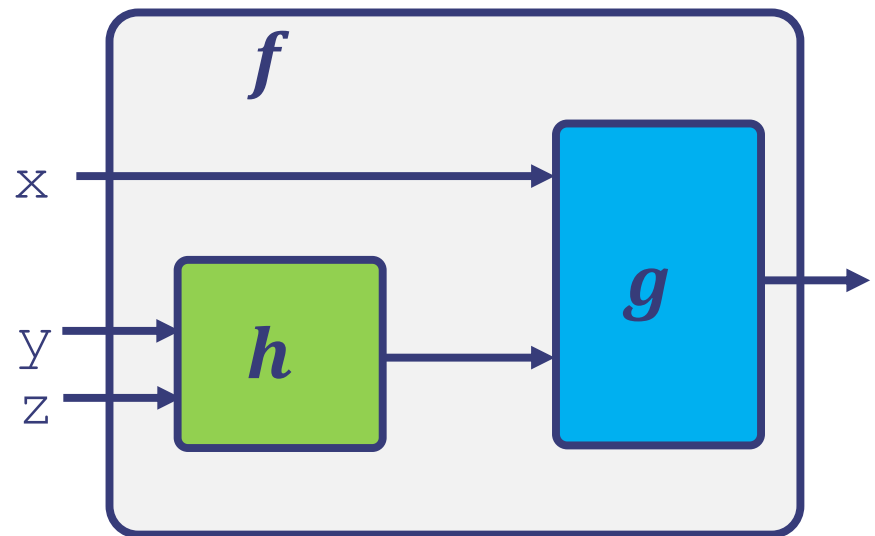
As we start to Attach things...



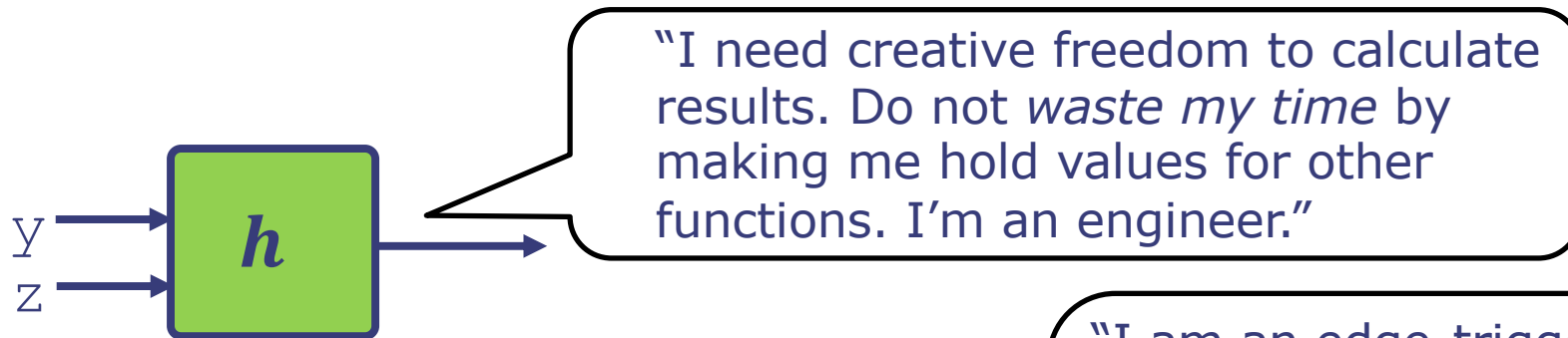
- Function g cannot start calculating its final answer until h has determined its own final answer.
 - If h needs t_{pdh} to calculate and g needs t_{pdg} that means...

System as a Whole?

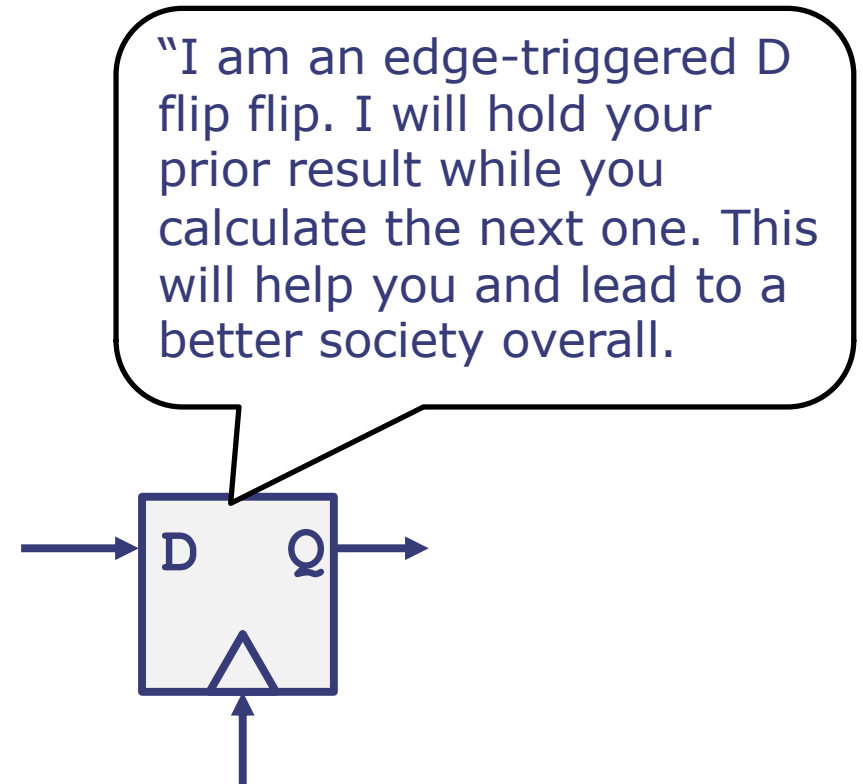
- The t_{pd} of the whole system has gone up!
- So the throughput of the whole system has gone down.
- But this is only half the problem.
- The module h is meant to calculate, but we're requiring it **to hold its result** for the benefit of g .
- That is a *waste* of its potential



How to Fix?

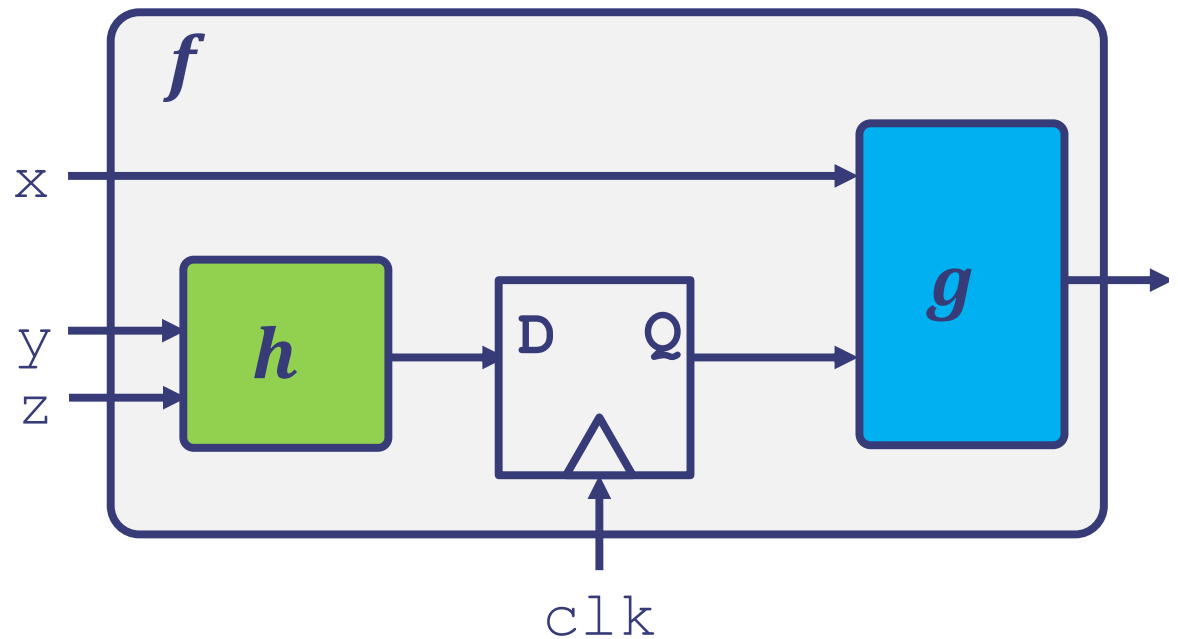


- We fix with flip flops



Fixing function f

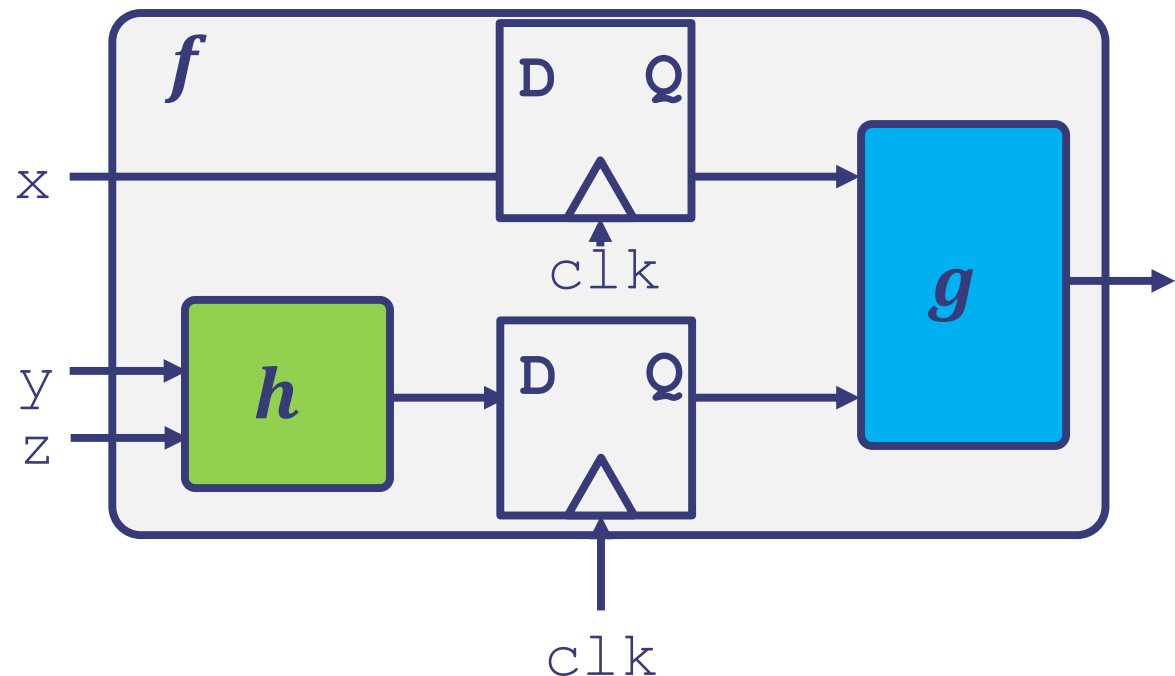
- Make clk a periodic signal
- Every so often grab a result from h and hold it input of g .
- Frees up h to calculate *at the same time* that g is calculating (albeit on different sets of data)



****review/reminder/spoilerz: this is potentially not perfect solution...it introduces other issues (we'll spend next few weeks talking about them)***

Fixing function f

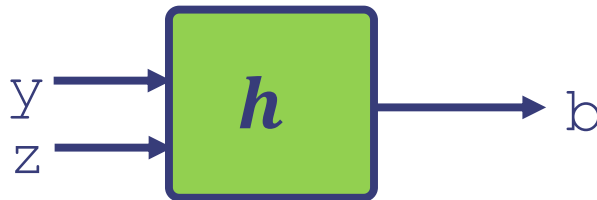
- We'll have to pipeline to non-slow path appropriately for $f(x, y, z) = g(x, h(y, z))$ to be more robust.
- And that will require continued thought/discussion.



Originally We had:

- Pure combinational version of the h function:

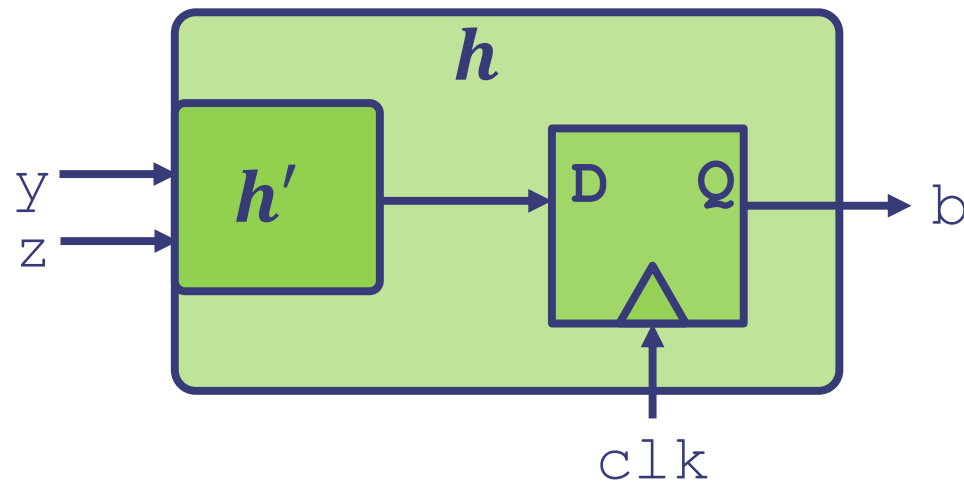
```
module h (input wire [7:0] y, input wire [7:0] z, output logic b);  
    always_comb begin  
        b = y>z;  
    end  
endmodule
```



Incorporate a Flip Flop in it.

- Same logic but put a flip flop on output

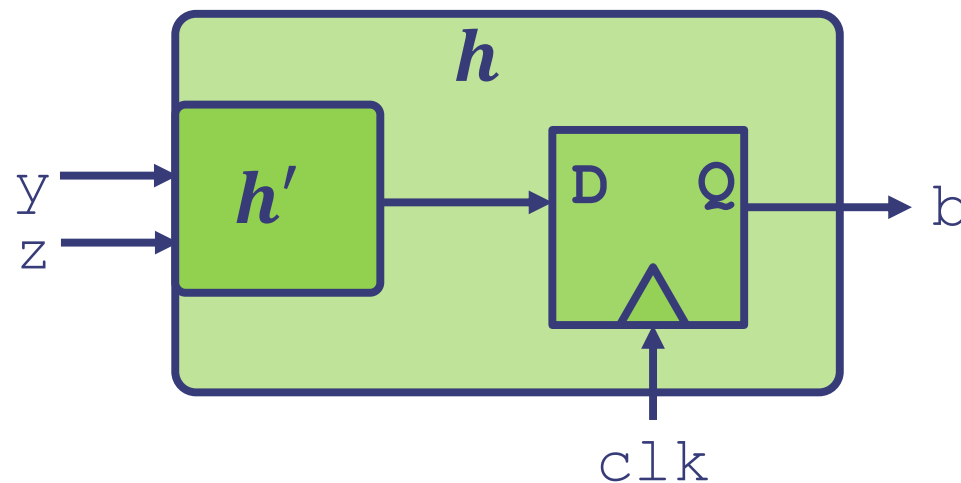
```
module h (input wire clk, input wire [7:0] y, input wire [7:0] z, output logic b);  
    logic b_i;  
  
    always_comb begin  
        b_i = y>z;  
    end  
  
    always_ff @(posedge clk) begin  
        b <= b_i;  
    end  
end  
endmodule
```



Incorporate a Flip Flop in it.

- Or said more succinctly...

```
module h (input wire clk, input wire [7:0] y, input wire [7:0] z, output logic b);  
    always_ff @(posedge clk) begin  
        b <= y>z;  
    end  
endmodule
```

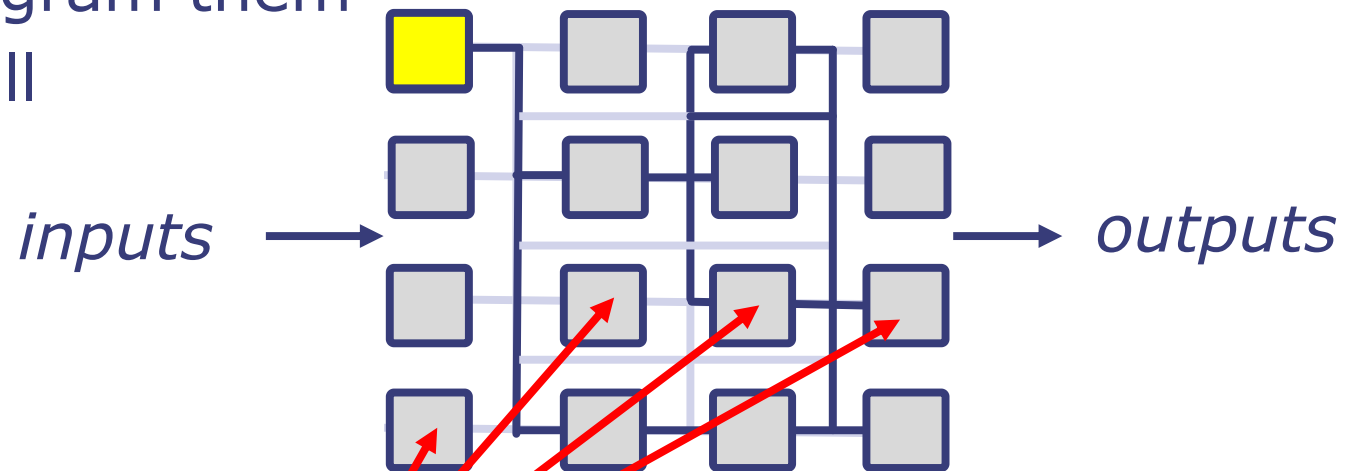


How do we know what t_{pd} and t_{cd} are in our design?

- Each 6-input programmable unit (LUT6) has a t_{pd} of about 150 picoseconds.
- So we could conceivably just figure out how many little logic functions each LUT6 uses and know our delays, right?
- ...not really

Remember...an FPGA

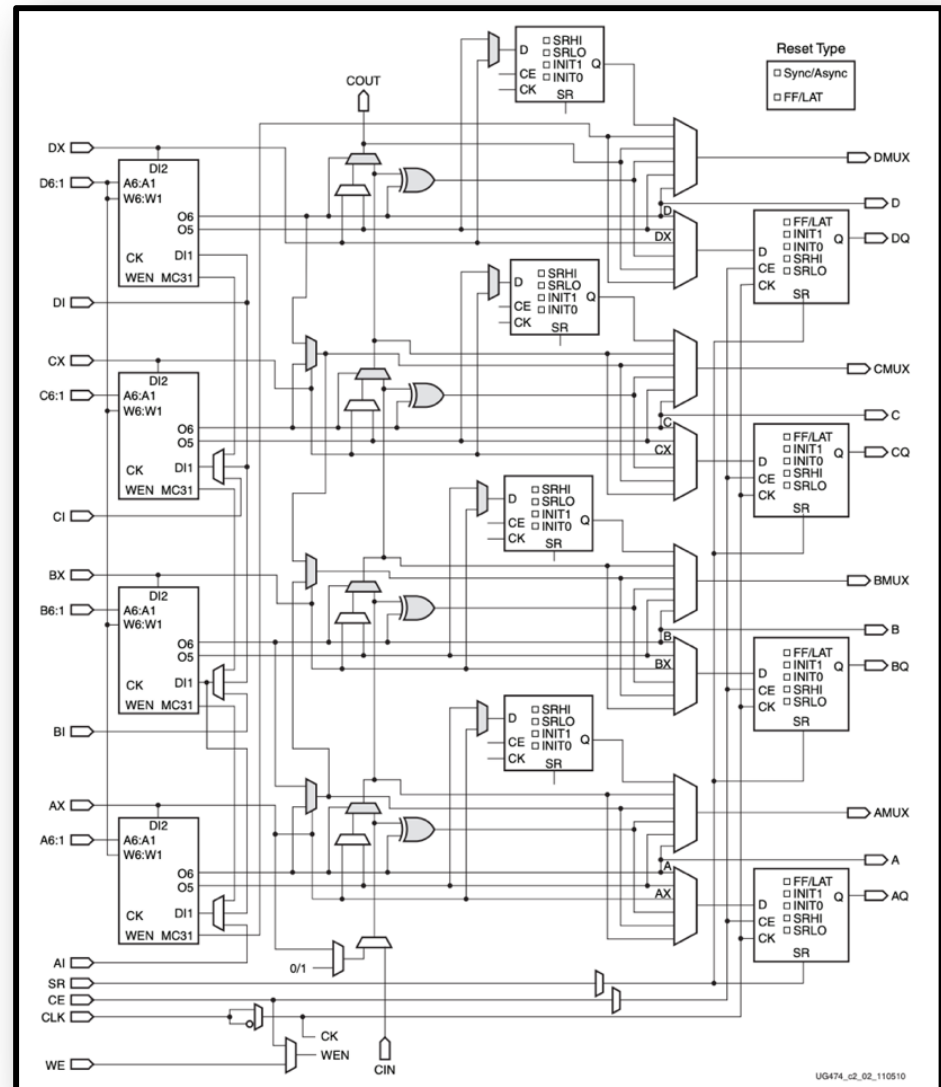
- Many small programmable blocks
- Write code for each and program them
- Write “code” that specifies how they work together and program them
- Run it all



Many small, relatively simple computational elements you individually program and individually connect together as you see fit!

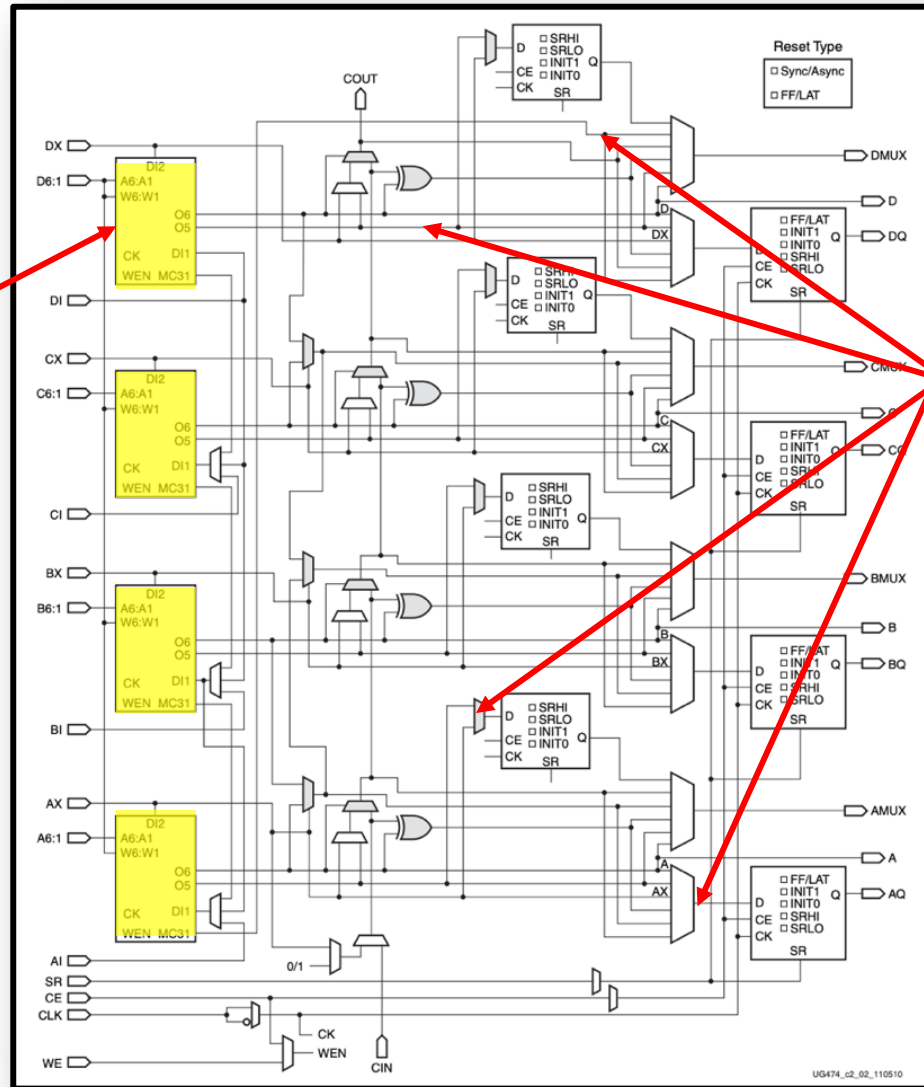
Xilinx Logic Blocks

- Our FPGAs have about 8500 of these →
- Called “Logic Slices”
- Each slice has four CLBs (“Configurable Logic Blocks”) that form the clay from which we sculpt our digital functions



Xilinx Logic Blocks

This is what has a t_{pd} of 150 ps



All the rest, both internal to this block and between blocks has its own delays

The Design Process

- Taken all together it is very difficult to by-eye know the timing of a particular design.
- We usually design (with best practices), build using a tool, have the tool analyze the result to give us the actual timing, and iterate on the design as needed.
- In some ways it was vibe-coding before vibe-coding was a thing.

One Last Thought for Day 1

- This is glimpse of the digital design situation.
 - It is a constant battle against time.
 - Worrying about how long an operation will take
 - Making sure no part of your design is wasting time
-
- Meeting timing in your designs will become a key piece of who *you* are as a person. It is how you will judge your own self-worth

What's Next

- Get Lab Kit (you can get in office hours starting after 4:15 pm -6:15pm today...or at any office hours with a TA...see calendar...so that's Subhi, Nerissa, Keilee, Heba, Yeabsira, or Joe)
- Week 1 assignments are out now (on course website). Due next Wednesday September 16 at night at 11:59pm