

# Open Source Digital Logic Design

Using open source tools for designing, testing, and synthesising  
Verilog modules

Davis Claiborne

LUG @ NC State

August 31, 2021



**Linux Users Group**  
at NC State University

# What is digital logic?

- Circuits need to do things
- Claude Shannon: *A Symbolic Analysis of Relay and Switching Circuits*
  - Representing circuits with Boolean algebra
  - Boolean: 0 or 1 (high/low voltage)



[11]

## └ Digital Logic

## └ Boolean algebra

## └ What is digital logic?

- Circuits need to do things
- Claude Shannon: *A Symbolic Analysis of Relay and Switching Circuits*
  - Representing circuits with Boolean algebra
  - Boolean: 0 or 1 (high/low voltage)



- Basically, digital logic is getting circuits to do things
- For a long time, getting circuits to do thing thing you wanted was more of an art than a science
- That was true until Claude Shannon proved that any circuit can be represented using Boolean algebra; previous study showed that any logic expression can be expressed with Boolean algebra

# What is Boolean algebra?

- Values can be 0/1, F/T, etc.
- Basic operations are “and,” “or,” and “not”
  - These are required to represent any expression
  - More operators exist<sup>1</sup>
- Can only answer yes or no questions
  - If you abstract the yes or no questions enough, can do math—more later

---


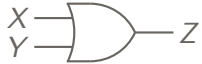

<sup>1</sup> And for some, a *single one* (e.g. NAND) can represent all logic operations!

- Values can be 0/1, F/T, etc.
- Basic operations are "and," "or," and "not"
  - These are required to represent any expression
  - More operators exist?
- Can only answer yes or no questions
  - If you abstract the yes or no questions enough, can do math—more later

<sup>1</sup> And for some, a single one (e.g. NAND) can represent all logic operators

- So what exactly is Boolean algebra?
- Basically, Boolean algebra is using two opposite states, usually represented as 0 and 1, true and false, or some other pair to represent logic expressions
- Using the basic operators and, or, and not, any logic expression can be represented
- Now, you may be wondering: what can Boolean algebra answer? What exactly is a logic expression?
- Boolean algebra can answer any question that can be answered strictly with “yes” or “no” questions
- Those of you more in the know know that last statement isn't totally true - I'll get back to that later, don't worry

# Notation

	Logic notation	EE expression	Logic gates
And	$Z = X \wedge Y$	$Z = X \cdot Y$	
Or	$Z = X \vee Y$	$Z = X + Y$	
Not	$\neg X$	$\bar{X}$ <sup>1</sup>	




<sup>1</sup> Sometimes you'll see people use a single apostrophe/tick as well, i.e.  $X'$

## Open Source Digital Logic Design

## └ Digital Logic

## └ Boolean algebra

## └ Notation

	Logic notation	EE expression	Logic gates
And	$Z = X \wedge Y$	$Z = X \cdot Y$	
Or	$Z = X \vee Y$	$Z = X + Y$	
Not	$\neg X$	$\bar{X}$	

\* Sometimes you'll see people use a single asterisk (\*) as well, i.e.  $X^*$

- There are three primary notations: formal logic notation, the electrical engineering expression form, and logic gate notation, which is also used by electrical engineers
- As a somewhat biased electrical/computer engineer, I much prefer the last two, just because those are what I'm used to seeing
- If you're wondering why ECE people use two representations, the reason is essentially just that the two different notations are good for different things; the expressions are much more compact and easier to optimize (generally), but logic gates are much easier to look at and follow visually

# Truth tables

And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	$\bar{X}$
0	1
1	0



And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	$\bar{X}$
0	1
1	0

- Truth tables are one way of analyzing the outputs of logical expressions
- They show the output for every permutation of inputs; the order of the inputs is traditionally done starting at all zeros and going to all 1s, alternating bits starting from the right until all permutations are seen
- Here I'm showing the truth tables for the 3 basic logic gates
- Reading them isn't too hard, just think of it as a true or false problem

# Truth tables

And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	$\bar{X}$
0	1
1	0

*Is X true AND is Y true? No*

└ Digital Logic

└ Boolean algebra

└ Truth tables

And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	X
0	1
1	0

*Is X true AND is Y true? No*

- For instance, here's a quick example: in the highlighted row of this truth table, are X and Y both true? The answer is no, since Y is 0 (false)

# Truth tables

And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	$\bar{X}$
0	1
1	0

*Is X true AND is Y true? Yes*

└ Digital Logic

└ Boolean algebra

└ Truth tables

And:

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Or:

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Not:

X	X
0	1
1	0

Is X true AND is Y true? Yes

- Here's another quick example: are X and Y both true now? The answer is yes, because both X and Y are 1
- So, why do we use truth tables?
- Basically, they're just another way to help look at how the inputs and outputs relate for a given input

# Finite State Machines

- Allow more complex, useful circuits
- Two main types:
  - **Moore**: Output depends only on state
  - **Mealey**: Output depends on state and inputs
- How to store state?

└ Digital Logic

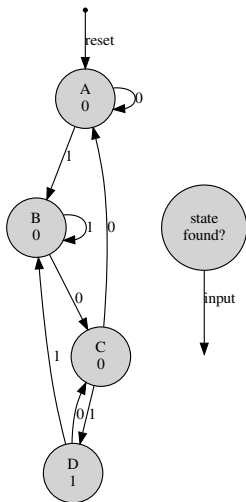
└ Finite State Machines

└ Finite State Machines

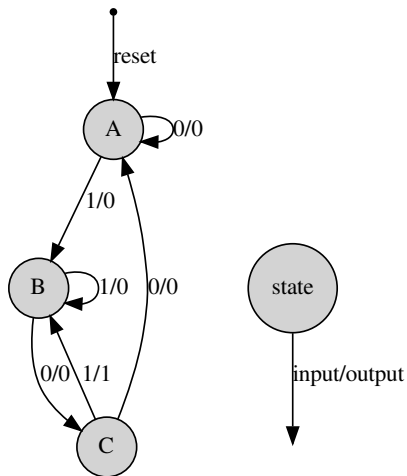
- Allow more complex, useful circuits
- Two main types:
  - Moore: Output depends only on state
  - Mealey: Output depends on state and inputs
- How to store state?

- Finite state machines are a way to make more complex, useful circuits by adding state information to them
- They're really more of a design tool, since all these things could be done without a state machine, it would just be a pain
- There are two main types, Moore and Mealey
- The main difference between the two is whether the output depends on the input or not: for Moore it depends only on the state, for Mealey it also depends on te inputs
- What this means is that Moore machines tend to be larger and simpler, while Mealey machines are smaller, but more complex

# Moore vs Mealey



Moore 101 detector



Mealey 101 detector

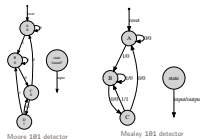


## Open Source Digital Logic Design

## └ Digital Logic

## └ Finite State Machines

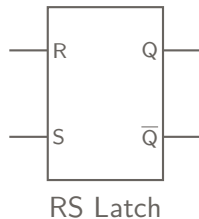
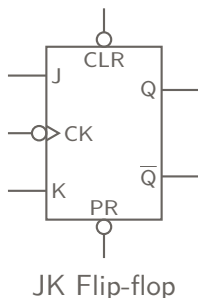
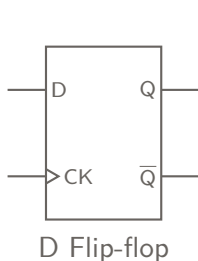
## └ Moore vs Mealey



- Here are two example FSMs where you can see the differences between Moore and Mealey more easily
- Both are looking to detect the input pattern “101”, but look fairly different
- Let’s start looking at the Moore FSM:
  - To the right of the graph you can see the general template of how to read this graph: each circle is a state, where the top line shows the name and the bottom line shows the output; arrows coming out of the circle represent an input causing the state to change
- Starting at A, since that where it gets reset to, it should be fairly easy to convince yourself this does, in fact, work
- Now, let’s look at the Mealey FSM:
  - In its case, the circle just represents a state, since Mealey FSM outputs rely in the state and the input, which is why the output is part of the arrow here
- One thing I’ve kind of glossed over so far, is how do you store information in digital logic? Not just in FSMs, but in general?

# Storage

- Flip-flops, latches, registers



- Triangle often represents clock
- Circle represents active low logic

## Open Source Digital Logic Design

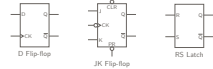
Digital Logic

Finite State Machines

Storage

## Storage

- Flip-flops, latches, registers



- Triangle often represents clock
- Circle represents active low logic

- Storage in circuits is done using what are known as flip-flops, latches, or registers
- The terminology you use depends on a lot of different things, but for the most part the terms are interchangeable enough at a *high level* that it doesn't really matter
- In reality, these elements all differ in their implementation and use, but I won't go too deep into them here
- I will point out a few general things for reading these, though:
- Note how the D and JK FF have an input called CK - that stands for "clock," and is often represented with the triangle on the pin
- The circle you see on some of the pins on the JK FF means that it uses "active low" logic

# Why optimize?

- Fewer components
  - Cheaper
  - Simpler
  - Faster
  - Efficient
  
- So how is it done?

# Open Source Digital Logic Design

## Digital Logic

### Optimization

#### Why optimize?

#### Why optimize?

- Fewer components
  - Cheaper
  - Simpler
  - Faster
  - Efficient
- So how is it done?

- One of the key benefits Claude Shannon brought to digital logic with the introduction of Boolean algebra was a reliable way to optimize the expressions down to simpler formats
- Optimized expressions offer a number of advantages; by reducing the number of components in a circuit, you can make it cheaper, simpler, faster, and more efficient
- Optimize circuits are less expensive because they have fewer components
- They're easier to debug because they're simpler, making it faster to find mistakes
- They're faster, because each component introduces propagation delay as the current travels through it
- They're more efficient because each component consumes power
- There are a number of techniques for performing optimization that I'll quickly discuss

# Optimization technique 1: Reduce logic expressions

- Theorems:

- $X + 0 = X$

- $X + 1 = 1$

- $X \cdot 0 = 0$

- $X \cdot 1 = X$

- $\overline{\overline{X}} = X$

- $X + X = X$

- $X + \overline{X} = 1$

- $X \cdot X = X$

- $X \cdot \overline{X} = 0$

- $X(Y + Z) = XY + XZ$

- $XY + XZ = XZ + XY$

- These can be used to help reduce expressions:

- $XY + \overline{X}Y = Y(X + \overline{X}) = Y \cdot 1 = Y$

## • Theorems:

- $X + 0 = X$
- $X + 1 = 1$
- $X \cdot 0 = 0$
- $X \cdot 1 = X$
- $\overline{\overline{X}} = X$
- $X \cdot \overline{X} = 0$

- $X + X = X$
- $X + \overline{X} = 1$
- $X \cdot X = X$
- $X \cdot \overline{X} = 0$

- $X(Y + Z) = XY + XZ$
- $XY + XZ = XZ + XY$

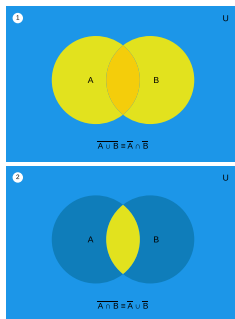
## • These can be used to help reduce expressions:

- $XY + \overline{X}Y = Y(X + \overline{X}) = Y \cdot 1 = Y$

- In Boolean algebra, there are a number of theorems that can be used to reduce logic expressions
- Here are some of the more simple theorems - I won't go through them all or prove them for the sake of brevity, but most of them can be proven fairly easily just by drawing the truth table
- You can reduce logic expressions using these theorems (in both directions) to eliminate redundant terms
- The main problem with this technique is that it requires a bit of practice for some of the longer, more complicated expressions, and it just isn't really all that efficient to do - lots of guess and check

## More theorems: De Morgan's Law

- $\overline{(A + B)} = \bar{A} \cdot \bar{B}$
- $\overline{(A \cdot B)} = \bar{A} + \bar{B}$
  
- “Break the line, change the sign”



[12]

$$\overline{(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C})} = \overline{\bar{A} + \bar{B} + \bar{C}} + \overline{\bar{A} + \bar{B} + \bar{C}} = \bar{\bar{A}} \bar{\bar{B}} \bar{\bar{C}} + \bar{\bar{A}} \bar{\bar{B}} \bar{\bar{C}} = ABC + ABC\bar{C} = AB(C + \bar{C}) = AB$$



## Open Source Digital Logic Design

## └ Digital Logic

## └ Optimization

## └ More theorems: De Morgan's Law

$$\bullet \overline{(A + B)} = \bar{A} \bar{B}$$

$$\bullet \overline{(A \cdot B)} = \bar{A} + \bar{B}$$

- "Break the line, change the sign"



$$\overline{\overline{A + B + C}} = \overline{\overline{A + B + C}} = A + B + C = \overline{\overline{A} \cdot \overline{\overline{B} + \overline{\overline{C}}}} = \overline{\overline{A} \cdot \overline{B + C}} = \overline{\overline{A} \cdot \overline{B} \cdot \overline{C}} = ABC + AB\bar{C} + A\bar{B}C + \bar{A}BC = AB$$

- One of the more advanced theorems I'll cover here is called De Morgan's Law, which lets you distribute inversion operations
- In my opinion, it's easier to visualize using set theory, which you can see on the right
- In the first part of the picture, you can read it as "everything that's not in A or B is the same as everything that's not in A combined with everything not in B"
- Similarly, in the bottom picture, you can read it as "everything not in the intersection of A and B is the same as everything not in A combined with everything not in B"
- A way to remember how to do it: "Break the line, change the sign"
- Here's a good example of why this is so useful: you *could* FOIL out those terms, find all the like terms, etc., but De Morgan's is so much easier and faster to do

## Optimization technique 2: K-maps

$X$	$Y$	$Z$
0	0	1
0	1	0
1	0	1
1	1	0

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

└ Digital Logic

└ Optimization

└ Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

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

- Karnaugh maps, usually called K-maps, are another tool that you can use to help reduce logic expressions
- The basic concept is to rearrange the truth table such that *only one of the inputs changes* as you change columns or rows, *even when you wrap around*
- That's a little jargony, so it's easier to look at an example; let's put the first truth table into a k-map

## Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X \ Y	0	1
0	1	1
1	0	0

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

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

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

- I've color coded the items to help show the transformation
- Reading these is a lot like consulting a table; for a given row or column, the specified value of X applies
- Now that the items are in the table, you can circle groups of contiguous 1s that are a power of two size and rectangular, maximizing the size of each grouping

## Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X \ Y	0	1
0	1	1
1	0	0

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$

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

## Open Source Digital Logic Design

## Digital Logic

## Optimization

## Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

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

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$

- For instance, here the first row has 2 1s in a rectangle; since 2 is a power of 2, you can group them
- Each grouping corresponds to a part of the final expression
- You can find that expression by looking at the columns and rows shared and doing some optimizations
- This may seem like optimization 1 with extra steps, but the main advantage of this is that it's fast, reliable, and easy to do
- The reason this works is precisely *because* of the way the table was layed out - by only changing one term at a time, it makes logic simplifications always reduce (when done properly)

## Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X \ Y	0	1
0	1	1
1	0	0

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$

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

BC \ A	0	1
00	0	1
01	0	0
11	1	1
10	1	1



X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X	0	1
Y	0	1
0	1	0
1	0	1

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$
  

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

BC	0	1
00	0	1
01	0	0
11	1	1
10	1	1

- Now let's look at a more complicated case. This one's a bit trickier to set up - notice how in the third row it goes from 01 to 11 in - this is so that only one term changes at once
- After filling in the table, we can do the same process as before: find rectangular power of 2-size groups of 1
- And remember: you can wrap "around" the top and bottom or left and right of the table, since it's still only one term that's changing, and you always want to go for the largest groupings possible

## Optimization technique 2: K-maps

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

	X	
Y	0	1
0	1	1
1	0	0

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$

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

	A	
BC	0	1
00	0	1
01	0	0
11	1	1
10	1	1

$$Z = \overline{B} + A\overline{C}$$

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	0

$$Z = \overline{Y}(\overline{X} + X) = \overline{Y}$$
  

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

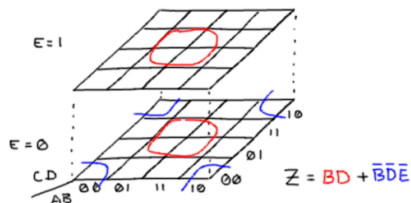
  

A	B	C	Z
00	0	0	0
01	0	0	0
11	1	1	1
10	0	0	1

$$Z = \overline{B} + A\overline{C}$$

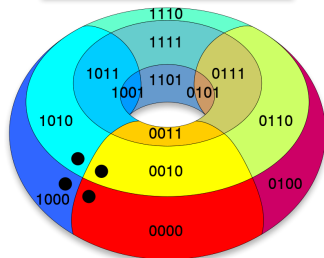
- Here's the final result for that grouping
- There's a few things to notice here:
- First, notice how the blue grouping wrapped around the top and bottom, becoming term  $A C'$
- As an aside, this is part of the reason why I think it's actually easier to think of this as a sphere, though that's harder to show
- Next, notice how larger groups are preferred: the red block could have been two distinct blocks of 2 each, but since it would simplify out anyways, it's easier to have a single block
- Finally, notice how  $ABC'$  is in two groups,  $B'$  and  $AC'$ . At first, covering it twice might seem like a waste. But counting it twice actually allows us to simplify: instead of  $AB'C'$  we can have just  $AC'$ .

# 3D K-maps: AKA when did this become topography?

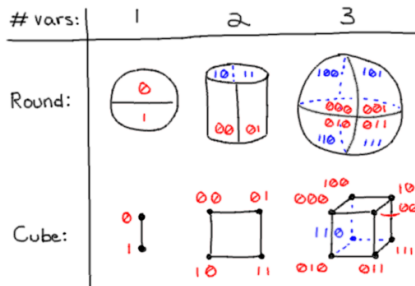


4 variables:

0000	0100	1100	1000
0001	0101	1101	1001
0011	0111	1111	1011
0010	0110	1110	1010



[13]



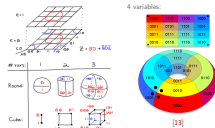
## Open Source Digital Logic Design

## Digital Logic

## Optimization

3D K-maps: AKA when did this become topography?

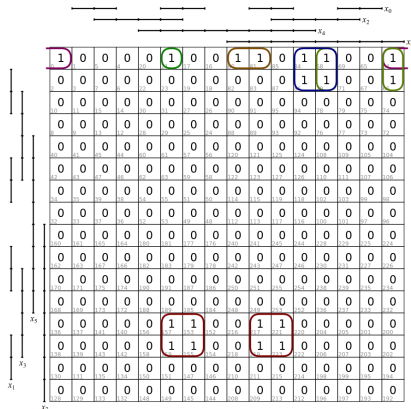
3D K-maps: AKA when did this become topography?



- Beyond 4 variables in a K-map, you have to do some fiddling around to get the topography to cooperate properly
- One of the simplest ways to do this is to overlay two matrices on top of each other
- If these maps get too much larger, you run into an issue, as you either won't get the most efficient possible expression anymore or you'll get an incorrect expression depending on how the terms are laid out
- This is another reason I prefer thinking of K-maps as curved shapes shapes like cylinders, spheres, and toruses, as they implicitly remind you of the limitations of these representations
- Excuse the mostly horrible images drawn by me... these things are well beyond my tikz-fu

# Beyond six variables: Try not to

8 variables<sup>1</sup>:



$$y = (x_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 x_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 x_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 x_5 x_4 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 \bar{x}_5 x_4 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 \bar{x}_5 x_4 x_3 \bar{x}_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 \bar{x}_5 x_4 x_3 x_2 \bar{x}_1 \bar{x}_0) \vee (x_7 \bar{x}_6 \bar{x}_5 x_4 x_3 x_2 x_1 \bar{x}_0) \vee (x_7 \bar{x}_6 \bar{x}_5 x_4 x_3 x_2 x_1 x_0)$$

[10]

<sup>1</sup> Technically, this is a Karnaugh-Veitch chart, but whatever

# Open Source Digital Logic Design

└ Digital Logic

└ Optimization

└ Beyond six variables: Try not to

Beyond six variables: Try not to

8 variables<sup>1</sup>:



[30]

<sup>1</sup> Technically, this is a 256-cell Karnaugh map, not a Karnaugh map.

- There is a limit to how useful a K-map can become; after a certain point, it becomes more academic than practical in my opinion
- All the mirroring, odd group sizes, etc., make this pretty tedious to do
- This about wraps up all that I really feel I need to cover for digital logic for now - there's a lot more that I *could* cover, but you're not prepping for the FE exam or anything, so I'll leave it here for now
- In case you're curious about how expressions with more than six variables are optimized...
- There are other tools to aid performing this process by hand, like Reduced Karnaugh maps, but it's mostly reserved for computers, who are much better at doing tedious things, both in terms of speed and error rate

## Specifying the circuit

- Need to specify logic circuit to computer
- “Compiling” (+synthesis) optimizes logic
- Hardware description languages:
  - Verilog
  - VHDL (VHSIC (Very High Speed Integrated Circuit) HDL)



- Need to specify logic circuit to computer
- "Compiling" (= synthesis) optimizes logic
- Hardware description languages:
  - Verilog
  - VHDL (VHSIC (Very High Speed Integrated Circuit) HDL)

- The first step into getting computers to do the optimization work for you is... telling the computer what your circuit is
- Parts of this process can be considered fairly similar to what a compiler is doing as it optimizes your code; this is usually done during the synthesis step
- There are a few major ways to specify your circuit, the most common being what are called "Hardware description languages," or HDLs, like Verilog and VHDL
- For this presentation, I'll talk about Verilog, just because I know the most about it, though both HDLs see wide use

# What is Verilog?

- HDL with C-like syntax
- Two parts of language:
  - **Synthesizable**: Can be represented as circuit
  - **Unsynthesizable**: Exists for logic verification
- **Non-linear** execution

- HDL with C-like syntax
- Two parts of language:
  - **Synthesizable**: Can be represented as circuit
  - **Unsynthesizable**: Exists for logic verification
- **Non-linear** execution

- Verilog is a hardware description language designed with a C-like syntax
- Verilog can be broken down into two key parts: synthesizable and unsynthesizable
- The synthesizable portion of Verilog is what's used to optimize the logic expressions mentioned earlier
- The unsynthesizable part is used to help test your circuits to ensure they're doing what you want
- One important thing to keep in mind when working with Verilog is that it's **not** a traditional programming language, because it does not run linearly
- This can be hard to wrap your head around at first, but it's important to remember this, because the code is describing a circuit, which (of course) does not run linearly

# Basic syntax

Terminology:

- **Module:** logic circuit; functions
- **Port:** inputs/outputs to module; parameters

```
module and_behavioral(  
    input wire a, b, // Input ports  
    output reg z     // Output ports  
);  
  
    // always @(): Reevaluate any time a or b changes  
    // Think of this like a while-true that only executes when its inputs change  
    always @( a or b ) begin  
        // 1'b1: one bit long number binary 1  
        // &&: logical and  
        // begin: {; end: }  
        if ( a == 1'b1 && b == 1'b1 ) begin  
            z = 1'b1;  
        end  
        else begin  
            z = 1'b0;  
        end  
    end  
  
end  
  
endmodule
```

Verilog

Syntax

Basic syntax

## Terminology:

- **Module:** logic circuit, functions
- **Port:** inputs/outputs to module, parameters

```

module and_behavioral()
  input wire a, b; // Input ports
  output reg z; // Output ports

  // Always @*: Re-evaluate any time a or b changes
  // Think of this like a while loop that only executes when its inputs change
  always @(*) z = a & b;
  // Always @: and not using master library
  // Always @: and
  // If a == 1'b1 & b == 1'b1 : begin
  //   z = 1'b1;
  // end
  // else begin
  //   z = 1'b0;
  // end
endmodule

```

- First, let's start off with some basic terminology:
- In Verilog, any time someone says “module,” you can think of it as if it were a function in a normal programming language
- Any time someone says “port,” they mean the inputs and outputs of a module, which can be thought of as the parameters to a function
- Now, let's start looking at the code itself
- This code creates a module called “and\_behavioral,” which has two inputs, a and b, and an output, z
- Notice that the output is a reg: confusingly, this is different from the register memory storage elements
- Right off the bat, I'm throwing a kind of tricky piece of syntax at you: “always @” statements are used to constantly reevaluated every time one of the inputs changes
- You can think of this like an infinite loop that only runs if one of its inputs has changed
- Inside this block, we have an if-else statement
- To read the conditional, it's important to know that 1'b1 is just Verilog's way of saying “a one-bit value of binary 1”
- One unusual part of Verilog's syntax is that it eschews the use of curly braces for “begin” and “end” instead
- If you're wondering about the difference between a wire or a reg, don't worry - I'll get to it

## Different styles of Verilog

**Gate-level:** Define circuit by basic logic gates; uses gate primitives

**Dataflow:** Define circuit by function; uses built-in operators

**Behavioral:** Define circuit by output; uses if/else/switch/etc.

```
// Port types in-module
module and_gatelevel( a, b, z );
    input wire a, b;
    output wire z;

    // Gate-level
    // Name of and gate is `u1`
    AND u1 ( z, a, b );

endmodule

// Port types as parameters
module and_dataflow(
    input wire a, b,
    output wire z
);

    // Dataflow
    assign z = a & b;

endmodule
```

```
// Port types as parameters
module and_behavioral(
    input wire a, b,
    output reg z
);

    // Behavioral
    always @( a or b ) begin
        if ( a == 1'b1 && b == 1'b1 ) begin
            z = 1'b1;
        end
        else begin
            z = 1'b0;
        end
    end

endmodule
```

Verilog

Syntax

Different styles of Verilog

## Different styles of Verilog

Gate-level: Define circuit by basic logic gates; uses gate primitives

Dataflow: Define circuit by function; uses built-in operators

Behavioral: Define circuit by output; uses if/else/switch/etc.

```

// Gate-level, in-module
module and_gatelevel (a, b, c);
  input wire a, b;
  output wire c;
  // Gate-level
  // Note: if/else gate has 'if'
  c = (a & b);
endmodule

// Gate-level, as parameters
module and_gatelevel (
  input wire a,
  input wire b
);
  // Behavioral
  always @(a or b) begin
    c = (a & b);
  end
endmodule

// Dataflow
module and_dataflow (
  input wire a,
  input wire b
);
  // Behavioral
  assign c = a & b;
endmodule

```

- Now that we have a basic understanding of Verilog's syntax, lets look at some of the different styles you can use
- First, take a look at `and_gatelevel`, which shows the gate-level style of Verilog
- This is the most intuitive for us EEs, since it breaks down the circuit into basic logic-gates
- Also note that this module is using in-module annotations; both versions are acceptable, but I prefer annotating the ports as parameters because I find it less prone to mistakes
- Up next is `and_dataflow`, which demonstrates the dataflow style of Verilog.
- Dataflow can be considered a step-above gate-level in terms of abstraction; by using the built-in operators you can abstract away thinking about low-level gates and instead focus on what the module is doing
- Finally, there's behavioral, the highest abstraction-level. I showed this first because it's the easiest for programmers to think of; by defining the module just in terms of inputs and outputs, you can abstract away virtually all parts of the circuit itself
- You may be asking yourself, "why bother with gate-level or dataflow when behavioral just does it all for you?" And that's a very fair question. Gate-level and dataflow are used today mostly for verification, since synthesis tools create efficient enough chips for the vast majority of cases
- Essentially, it can be considered writing C/Java/etc. vs assembly

# Testing Verilog: Test benches

```

module and_tb; // _tb = test bench
    reg a, b, // AND inputs
        check; // Bit toggled to trigger check
    wire z_dataflow, z_behavioral; // Outputs

    // Creates modules to test
    // .<x>(<y>) specifies that <y> should be assigned to port <x>
    // Typically, module being tested is called DUT
    and_dataflow add1( .a( a ), .b( b ), .z( z_dataflow ) );
    and_behavioral add2( .a( a ), .b( b ), .z( z_behavioral ) );

    // Shows outputs for manual checking
    // More sophisticated checking methods exist
    always @( check ) begin
        $display(
            "Time: %d a: %b b: %b z df: %b z bh: %b",
            $time, a, b, z_dataflow, z_behavioral
        );
    end

    // Changes values
    initial begin
        // #<x> = Go forward <x> clock cycles; ~ = toggle bits
        #1 a = 0; b = 0; check = 0; // Time: 1 a: 0 b: 0 z df: 0 z bh: 0
        #1 a = 0; b = 1; check = ~check; // Time: 2 a: 0 b: 1 z df: 0 z bh: 0
        #1 a = 1; b = 0; check = ~check; // Time: 3 a: 1 b: 0 z df: 0 z bh: 0
        #1 a = 1; b = 1; check = ~check; // Time: 4 a: 1 b: 1 z df: 1 z bh: 1
    end
endmodule

```



```

module add1s1 // 2's + 1's test bench
    #100 // 100 ns delay
    check; // 100 ns delay for integer check
    wire a,b,c,din,dout; // Signals

    // Create modules to test
    // combinatorial operations that you should be assigned to port 'a'
    // typically, module being tested is called 'DUT'
    and2to1to4out1 out1 (a [1], a[0], 1, out1[3:0]); // 1
    and2to1to4out2 out2 (a [1], a[0], 0, out2[3:0]); // 0

    // Show results for second checking
    // Here, unimplemented checking methods exist
    always @ (posedge clk)
        $display("Time: %d, a: %d, b: %d, c: %d, d: %d, din: %d",
            $time, a, b, c,din,out1[3:0],out2[3:0]);

end

// Changes values
initial begin
    // How to do forward and check output -- a toggle logic
    #10 a = 0; b = 0; c = 0; // Time: 0, a: 0, b: 0, c: 0, din: 0, out1: 0, out2: 0
    #10 a = 1; b = 0; c = 0; // Time: 1, a: 1, b: 0, c: 0, din: 0, out1: 1, out2: 0
    #10 a = 0; b = 1; c = 0; // Time: 2, a: 0, b: 1, c: 0, din: 0, out1: 0, out2: 1
    #10 a = 1; b = 1; c = 0; // Time: 3, a: 1, b: 1, c: 0, din: 0, out1: 1, out2: 1
end
endmodule

```

- Now that we've written our Verilog module, we need a way to ensure that we've written it correctly
- Verilog does this using what's known as a test-bench
- Here you can see the basic layout of a testbench - it's actually very similar to a module
- That's because it is a module, just without any ports/I/O
- Here we create 3 regs: a and b, to hold the inputs to the adders, and check, which we'll use to trigger the output function
- We also create two wires, one for each of the outputs
- Next, the two modules are instantiated, add1 and add2, which are the dataflow and behavioral versions of the modules we created earlier
- Traditionally, the module being tested will be given the name DUT, but since two modules are being tested here, I can't do that
- Next, the always@ statement triggers output showing the current state of the circuit when check changes
- Finally, we have the code responsible for changing the values

# Automated test benches

```
module and_tb_selfchecking;

    reg [1:0] inputs [3:0]; // 4 inputs, 2 bits each
    reg outputs [3:0];      // 4 outputs, 1 bit each
    reg a, b, clk, reset;   // clk and reset are to prevent sim issues
    wire z_dataflow, z_behavioral;
    integer i, errors; // index for inputs/outputs, # errors

    // Creates modules to test
    and_dataflow add2( .a( a ), .b( b ), .z( z_dataflow ) );
    and_behavioral add3( .a( a ), .b( b ), .z( z_behavioral ) );

    // Pulses clock
    always begin #5; clk = ~clk; end

    // Initializes values
    initial begin
        inputs[0] = 2'b00; inputs[1] = 2'b01; inputs[2] = 2'b10; inputs[3] = 2'b11;
        outputs[0] = 1'b0; outputs[1] = 1'b0; outputs[2] = 1'b0; outputs[3] = 1'b1;
        errors = 0; i = 0; clk = 0;
        reset = 1; #1 reset = 0;
    end

    end

    // Assigns inputs; posedge means to only do it on the rising edge
    // (only when it changes from 0 to 1)
    always @( posedge clk ) begin
        #1 { a, b } = inputs[i]; // 'Unpacks' inputs[i] into a and b
    end

    end

    // Continued
```

# Open Source Digital Logic Design

└ Verilog

└ Testing

└ Automated test benches

## Automated test benches

```
module aut_32_addFunction;
    //sig [31]: output [32-bit] // A output, B data each
    //sig output [32-bit] // A output, B sig each
    //sig in [32-bit], result // sig and result are the previous sig values
    wire a_32of32in, a_32of32out;
    output [32-bit] result; // data for inputs/outputs, A errors

    // Creates module to test
    and_32of32in_and32 out = 1, in [31], in_0_32of32in [31];
    and_32of32out_and32 out = 1, in [31], in_0_32of32out [31];

    // Returns 32-bit
    always begin #1; out = result; end

    // Initialization values
    initial begin
        in[31] = 1'0; in[30] = 1'0; in[29] = 1'0; in[28] = 1'0; in[27] = 1'0;
        in[26] = 1'0; in[25] = 1'0; in[24] = 1'0; in[23] = 1'0; in[22] = 1'0;
        in[21] = 1'0; in[20] = 1'0;
        in[19] = 1'0; in[18] = 1'0;
        in[17] = 1'0; in[16] = 1'0;
        in[15] = 1'0; in[14] = 1'0;
        in[13] = 1'0; in[12] = 1'0;
        in[11] = 1'0; in[10] = 1'0;
        in[9] = 1'0; in[8] = 1'0;
        in[7] = 1'0; in[6] = 1'0;
        in[5] = 1'0; in[4] = 1'0;
        in[3] = 1'0; in[2] = 1'0;
        in[1] = 1'0; in[0] = 1'0;
    end

    // Design function, purpose seems to only do it on the rising edge
    // Only when the design runs to test it
    always @ (posedge in_0_32of32in) begin
        out = 1'0;
    end

    // Done
end
```

- There is a way to automate test benches so that you don't need to manually check if your outputs are right
- However, it's much more complex, and very prone to simulation issues caused by the non-sequential code operation
- Basically, some values in Verilog will change before others in an indeterminate manner, so you need to carefully design your tests around this fact
- I won't spend a ton of time going over how this is constructed, since it's a bit complicated, but I just figured I'd mention that it is possible to do

## Automated test benches (cont)

```

always @( negedge clk ) begin
    if ( ~reset ) begin
        // !==: Special kind of compare; doesn't ignore X's (don't cares)
        if ( z_dataflow !== outputs[i] ) begin
            $display(
                "dataflow  %d failed: %d vs %d",
                i, z_dataflow, outputs[i]
            );
            errors = errors + 1;
        end
        if ( z_behavioral !== outputs[i] ) begin
            $display(
                "behavioral %d failed: %d vs %d",
                i, z_behavioral, outputs[i]
            );
            errors = errors + 1;
        end
    end

    // Unless we have a test input of don't cares (which would be
    // pretty unusual), this is a fine way to see if we're at the end
    i = i + 1;
    if ( inputs[i] === 2'bx ) begin
        $display( "Test completed with %d errors", errors );
        $finish;
    end
end

end

endmodule

```

## Open Source Digital Logic Design

└ Verilog

└ Testing

└ Automated test benches (cont)

Automated test benches (cont)

```
always @(reset or clk) begin
  if (!reset) begin
    // If we're not doing off-chip, then we ignore it's (don't care)
    if (!x_external) begin
      #ns10;
      x_external = 1'b1;
    end
  end
  #ns10;
  if (!x_external) begin
    // If we're not doing off-chip, then we ignore it's (don't care)
    if (!x_external) begin
      #ns10;
      x_external = 1'b1;
    end
  end
end

// Makes sure we have a load of 1's on the output (which could be
// pretty common), but in a way that we can't see it on the
// I/O.
if (!x_external) begin
  #ns10;
  x_external = 1'b1;
end
end
end

endmodule
```

- More of the automated testing. I can go over this more at the end if people are interested and there's time
- A lot of the differences have to do with HDLs vs programming languages, and how you don't have control over when certain expressions are evaluated
- There's a lot of Verilog I haven't even covered, like blocking vs non-blocking, registers vs wires, and the finer points of Verilog's number representation and expansion, but I think this is enough for now

# Simulating Verilog—Option 1: Icarus

Website [6]

Basic usage:

```
$ iverilog -o out -s top file.v
```

- -o: Output executable; run this to simulate
- -s: Top-level module to simulate (or all if excluded)

```
$ vvp out
```

- Performs simulation

Icarus Verilog



└ Software

└ Simulation/Debugging

└ Simulating Verilog—Option 1: Icarus

Website [6]

Basic usage:

```
$ iverilog -o out -s top #file.v
```

- -o: Output executable; run this to simulate
- -s: Top-level module to simulate (or all if excluded)

```
$ vvp out
```

- Performs simulation



- Now that we've written all this Verilog code, it's time to actually simulate it
- Tons of simulators exist, but most of them are closed-source, or proprietary
- One that isn't, however, is called "Icarus Verilog"
- I won't go over installing it here, but I will give a quick overview of how to use it
- (See slide)
- There's a few more parts to this that I'll talk about later, but this is the gist of it

## Simulating Verilog—Option 2: CVC

- Website [2]
- Compiles Verilog to x86
- Vs Icarus Verilog:
  - Reportedly faster
  - Natively supports writing matrices to wave files<sup>1</sup>
  - Less-widely used
  - Documentation is fairly sparse<sup>2</sup>
- **Icarus**: Create VHDL output; preprocessors
- **CVC**: Explicit optimization control

---

<sup>1</sup> <https://stackoverflow.com/a/22395232/2238176>

<sup>2</sup> <https://raw.githubusercontent.com/cambridgehackers/open-src-cvc/master/doc/oss-cvc-quick-start-061014.pdf>



- Website [2]
- Compiles Verilog to x86
- Vs Icarus Verilog:
  - Reportedly faster
  - Natively supports writing matrices to wave files<sup>1</sup>
  - Less-widely used
  - Documentation is fairly sparse<sup>2</sup>
- Icarus: Create VHDL output; preprocessors
- CVC: Explicit optimization control

<sup>1</sup> <https://www.icarus.com/~steve/verilog-to-wave/>  
<sup>2</sup> <https://www.github.com/verilog-to-wave/verilog-to-wave/blob/master/README.md>  
<http://www.cvc1.com/~vstakhov/201202.pdf>

- Another option for simulating Verilog is CVC
- CVC compiles your verilog to x86 instructions to simulate, which reportedly improves speed, though I didn't bother checking this myself
- Another advantage of CVC over Icarus is that it can write matrix variables to wavefiles without having to do a workaround
- There are some downsides, however:
  - CVC is less-widely used than Icarus, making support for it harder to find
  - Documentation in general is pretty sparse: their website itself doesn't even have the documentation; it apparently comes with the code, but can also be found mirrored on GitHub
- Finally, each simulator has some different options
- Icarus can convert your Verilog to VHDL, and can also extend Verilog's functionality by adding preprocessor
- While Icarus does optimization, CVC gives you more explicit control of which optimizations to do

# GTKWave

Waveform viewer

Export wavefiles:

```
initial begin
    $dumpfile( "waves.vcd" );
    $dumpvars; // write all vars
end
```

More efficient wavefiles:

```
$ vvp out -fst
```

CVC-specific:

```
initial begin
    $fstDumpfile( "waves.fst" );
    $fstDumpvars; // write all vars
end
```

```
$ cvc +dump_arrays and.v
```

GTKWave's *incredible* logo



*An experienced professional shown violating most known rules of electrical safety with GTKWave. Please do not attempt this at home.*

[6]

# Open Source Digital Logic Design

Software

Simulation/Debugging

GTKWave

## GTKWave

Waveform viewer

Export wavefiles:

```
initial begin
  $dumpfile("wave.vcd");
  $dumpvars; // write all vars
end
```

More efficient wavefiles:

\$ vvp out -fst

CVC-specific:

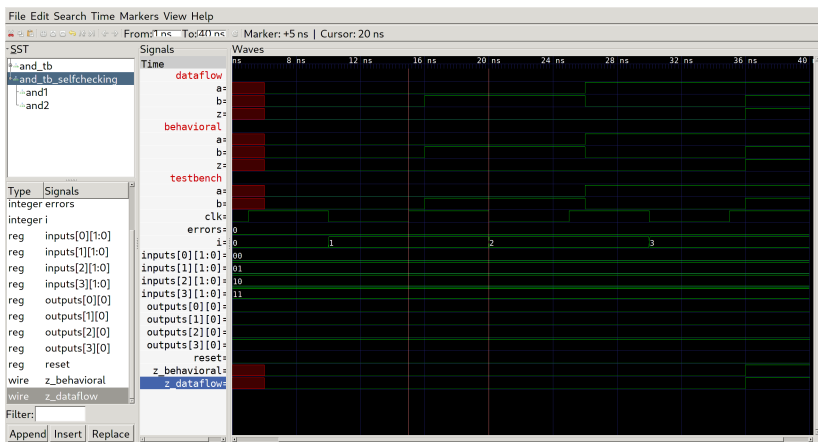
```
initial begin
  $dumpfile("wave.fst");
  $dumpvars; // write all vars
end
```

\$ cvc rdump\_arrays and.v



- GTKWave is a free, open-source waveform viewer with one of the best logos I've ever seen
- In case you don't know what a waveform viewer is, it's basically just a graphical way to view the relationship between signals
- Verilog has builtin commands for exporting wave files: `$dumpfile` specifies the file to write to, and `$dumpvars` specifies which variables to write (writes all by default)
- Additionally, most simulators offer support for more efficient wavefiles
- For instance, with Icarus, using `vvp` with the `-fst` flag to specify that wavefiles should be fsts
- For CVC, you need to explicitly use their fst-specific commands; the `dump_arrays` option includes arrays in wavefiles

# GTKWave in action



2021-08-31

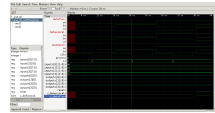
# Open Source Digital Logic Design

└ Software

└ Simulation/Debugging

└ GTKWave in action

GTKWave in action



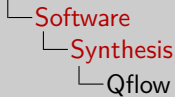
- Here you can GTKWave viewing the dumped variables from the self-checking testbench
- The idea is basically to mimic an oscilloscope
- It's fairly intuitive and easy to use, without any real "gotchas" in my opinion, so I won't spend too long on it

# Qflow

- Open source “digital synthesis flow”
- Most problematic tool
  - May just be my lack of experience
- Combines other tools:
  - yosys: Verilog synthesis/optimization
  - graywolf: Placement tool
  - qrouter: Routing tool



Qflow [7]



- Open source "digital synthesis flow"
- Most problematic tool
  - May just be my lack of experience
- Combines other tools:
  - yosys: Verilog synthesis/optimization
  - graywolf: Placement tool
  - qrouter: Routing tool



Qflow [7]

- Now onto synthesis - which is the process of turning your module into a real circuit
- The best Open Source synthesis tool I've found so far is Qflow
- That being said, I still run into a lot of errors with this tool, so I'd definitely say it's the biggest problem area
- However, it is also the tool I have the least experience with, since I normally have to use closed-source versions of tools for my classes
- From my understanding, Qflow is basically a wrapper around a series of other open source tools, like yosys, graywolf, qrouter and more in an attempt to automate the process

## Installation issues

**Note:** These issues were present as of August 31, 2021; some of these issues will likely be fixed upstream in the coming months

- **Graywolf:** Compilation issue due to globals<sup>1</sup>
- **netgen:** Compilation issue due to security flag<sup>2</sup>
- **Qflow:** Bad ABC symlink<sup>3</sup>

---

<sup>1</sup> <https://github.com/rubund/graywolf/issues/43>

<sup>2</sup> <https://aur.archlinux.org/packages/netgen-lvs-git/#comment-822826>

<sup>3</sup> <https://stackoverflow.com/questions/36593907/>



└ Software

└ Synthesis

└ Installation issues

**Note:** These issues were present as of August 31, 2021; some of these issues will likely be fixed upstream in the coming months

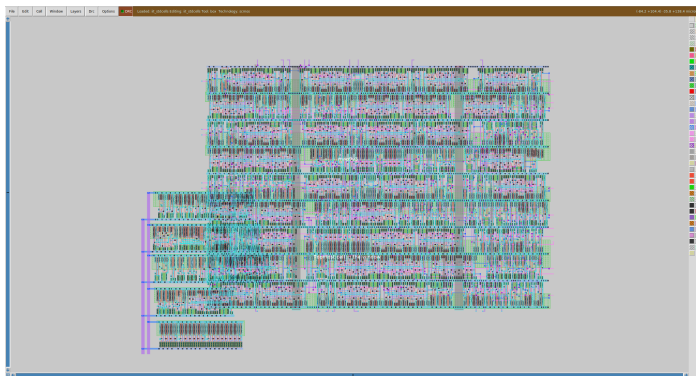
- **Graywolf:** Compilation issue due to `global`<sup>1</sup>
- **netgen:** Compilation issue due to security flag<sup>2</sup>
- **Qflow:** Bad ABC symlink<sup>3</sup>

<sup>1</sup> <https://github.com/ibland/graywolf/issues/43>  
<sup>2</sup> <https://www.ambition.org/netgen/netgen: too.glib.h: warning: #pragma GCC>  
<sup>3</sup> <https://stackoverflow.com/questions/5920867>

- While installing Qflow and its associated packages, I ran into a few errors that you should be aware of if you want to try this yourself
- Hopefully, these issues will be fixed in the coming months; I found two of them and their fixes myself, and plan on reporting them once I have some more information
- The first issue is with Graywolf; some global values are defined twice; the temporary fix involves just updating the makefile to resolve the issue
- The next issue I ran into was with installing netgen; it was also a compilation issue, where the temporary fix was also just updating the makefile
- The final issue I ran into during installation was with Qflow, which apparently assumes the location of ABC, so it creates a symlink to the wrong location. This can be fixed simply by figuring out where ABC actually installed and updating the symlink.

# Results

Results from following tutorial<sup>1</sup>:



If your design is too simple, you'll run into errors<sup>2</sup>

<sup>1</sup> <http://opencircuitdesign.com/qflow/tutorial.html>

<sup>2</sup> <http://opencircuitdesign.com/pipermail/eda-dev/2021-August/thread.html>

└ Software

└ Synthesis

└ Results

Results from following tutorial<sup>1</sup>:If your design is too simple, you'll run into errors<sup>2</sup>

<sup>1</sup> <http://openocm.com/design/2019/08/22/qflow-tutorial.html>  
<sup>2</sup> <http://openocm.com/design/2019/08/22/qflow-tutorial.html#2019-08-22-qflow-tutorial.html>

- Once you've got Qflow installed, you can follow the tutorial to see what the chip would look like
- You should not that it's not just a fire and forget process, as some parameters will need to be tweaked depending on what you're doing
- For instance, while trying to synthesize the simple and module we've been using through this presentation, I ran into errors
- This basically happened because the design was so small that it couldn't place power or ground properly. So just keep in mind that you will need to tweak some things to get it to work.

## More software/resources

In no particular order:

- **OpenROAD**: Attempt at fully automated EDA [9]
- **Verilator**: Compiles Verilog to C++; considered using it, but tests need to also be done in C++ [8]
- **Electric**: Integrated-Circuit design tool [5]
- **Circuit\_macros**: Tool for producing typeset schematics [1]
- **Digital**: Java-based tool for designing and simulating digital logic [3]
- **DigitalJS**: Online interactive Verilog tool [4]

# Open Source Digital Logic Design

Software

Synthesis

More software/resources

In no particular order:

- [OpenROAD](#): Attempt at fully automated EDA [9]
- [Verilator](#): Compiles Verilog to C++; considered using it, but tests need to also be done in C++ [8]
- [Electric](#): Integrated-Circuit design tool [5]
- [Circuit\\_macros](#): Tool for producing typeset schematics [1]
- [Digital](#): Java-based tool for designing and simulating digital logic [3]
- [DigitalJS](#): Online interactive Verilog tool [4]

- Of course, the software I've shown off here isn't the only software that exists
- There are a bunch of open source tools that I didn't show off, or just don't know about or use
- The first one, OpenROAD, from what I can tell is an attempt to make a fully automated HDL to hardware pipeline; I have not tried it out yet, but intend to give it a try soon
- Verilator is a Verilog simulator that simulates by converting the Verilog to C++; I considered using this, and it does look interesting, but to use it you need to write your tests in C++, since it only supports converting the synthesizable subset of Verilog; I may check it out in the future, but for now CVC works well enough
- Electric is an entire suite for designing ICs; I hadn't heard about it until recently, but it looks pretty neat; from what I can tell, it doesn't support synthesizing from Verilog, but it looks like a useful tool for other hardware design needs
- Circuit\_macros is the tool I used to produce all the logic gates for this presentation; it's basically like graphviz for circuits, although not quite as automated as I'd like it to be
- Digital and DigitalJS are both interactive tools for designing digital logic; Digital includes logic optimization tools, while DigitalJS runs in-browser and can read Verilog, which can be useful for debugging modules

# References I

## Software

---

- [1] Circuit\_macros—Typesetting tool for producing schematics  
[https://ece.uwaterloo.ca/~aplevich/Circuit\\_macros/](https://ece.uwaterloo.ca/~aplevich/Circuit_macros/)
- [2] CVC—Verilog simulator that compiles to x86 for speed  
<http://www.tachyon-da.com/what-is-cvc/>
- [3] Digital—Tool for digital logic design/simulation  
<https://github.com/hneemann/Digital>
- [4] DigitalJS—Online tool for viewing Verilog  
<http://digitaljs.tilk.eu/>
- [5] Electric—Integrated-Circuit Design system  
<https://www.staticfreesoft.com/index.html>
- [6] gEDA Projects—Collection of GPLd EDA tools, including Icarus Verilog and GTKWave  
<http://www.geda-project.org/>
- [7] Open Circuit Design—Collection of open-source EDA tools, including Qflow  
<http://opencircuitdesign.com/>
- [8] Verilator—Verilog transpiler to C++  
<https://www.veripool.org/verilator/>
- [9] OpenROAD—Goal is to make fully automated HDL to hardware pipeline  
<https://theopenroadproject.org/>

## Websites

---

- [10] Karnaugh-Veitch Map—Performs groupings for 4 to 8 variable k-maps  
<https://www.mathematik.uni-marburg.de/~thormae/lectures/t11/code/karnaughmap/>

## Images

---

# References II

---

- [11] Claude Shannon  
[https://en.wikipedia.org/wiki/Claude\\_Shannon#/media/File:ClaudeShannon\\_MF03807.jpg](https://en.wikipedia.org/wiki/Claude_Shannon#/media/File:ClaudeShannon_MF03807.jpg)
- [12] De Morgan's law  
<https://upload.wikimedia.org/wikipedia/commons/0/06/Demorganlaws.svg>
- [13] K-map on a torus  
[https://upload.wikimedia.org/wikipedia/commons/3/33/Karnaugh\\_map\\_torus.svg](https://upload.wikimedia.org/wikipedia/commons/3/33/Karnaugh_map_torus.svg)