Circuits

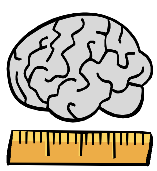
Part 5

Defining Boolean Logic

"Want to define it?"
"True."

Defining Boolean Logic

- Let's look at what exactly Boolean logic is in context of data types and functions
- Once we define the Boolean Data type, we can apply it to other systems

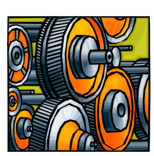


Boolean Logic and Sets

- Boolean values only have two possible values: True and False
- So, the set of values can be specified as {True, False} or, alternatively, as {1, 0}

Functions

- Also recall functions from earlier
- An *abstract data type* is a set of values and functions on those values
- So, we can define the data type for Boolean values



Defining Boolean Algebra

```
S = {0, 1}

* : S,S → S        "And"
+ : S,S → S        "Or"
' : S,S → S        "Negation"

For all x, y, z ∈ S the following is
true...
```

1. Associative

```
(x + y) + z  =  x + (y + z)

(x * y) * z  =  x * (y * z)
```


2. Commutative

```
x + y = y + x

x * y = y * x
```


3. Distributive

```
x * (y + z)  =  (x * y) + (x * z)

x + (y * z)  =  (x + y) * (x + z)
```


4. Identity

```
x * 1 = x

x + 0 = x
```


5. Complement

```
x * x' = 0

x + x' = 1
```


Extending Boolean to Other Types

- The Boolean Data Type can be written with these 5 properties: B = {`S,+,*,',0,1`}
- If we can show that some other type is a "Boolean algebra" if these five properties hold for all elements in S.

Example

```
Given a set U:

   S  =  P(U)
   0  =  { }
   1  =  U
   +  =  set union
   *  =  set intersection
   '  =  set negation
```


Example

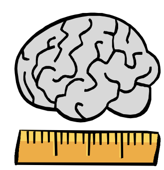
```
All we need to show is that:

(X ∪ Y) ∪ Z = X ∪ (Y ∪ Z)

...etc

which is what we observed with sets.
```


Boolean Algebra & Logic

From many, one

Boolean Algebra & Logic

- Boolean Algebra can be extended to other areas
- A subset of propositional logic can be put into the form of Boolean algebra



Propositional Logic in Boolean

```
S  =  {T, F}
1  =  T
0  =  F
*  =  ∧
+  =  ∨
'  =  ¬
```


Applying the Five Laws

- The five laws, stated before, can be applied to propositional logic
- So, at a stroke, this gives us a very rich environment in which we can manipulate logic propositions
- So, we can treat logic as algebra

And, Or, Not

- All we need is: And, Or, and Not
- This is because, implications and equivalences can be expressed with them.

$$a \rightarrow b \quad = \quad \neg a \lor b$$

$$a \leftrightarrow b \quad = \quad a \rightarrow b \land b \rightarrow a$$

Circuits

Boolean Hardware

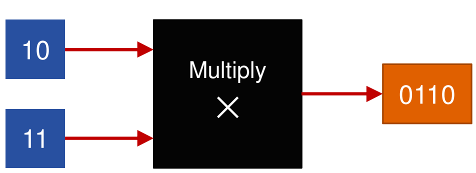
Circuits

- We use Boolean algebra because it can represent logical functions
- Electronic devices use logic to do their computation

Circuits

- Boolean algebra gives designers tools to design & analyze solutions
- We can now look at designing electronic circuits to make simple computations

Two Bit Multiplier? Can we make it?



Designing It

- To design a circuit that multiplies two 2-bit numbers, we can use *Boolean algebra*
- We need to figure the logic – given that bits of 1 and 0 will map directly to truth values
- The result of the algebra will be the desired output

It Takes the Following Skills

1. Design a truth-table to represent the different inputs and the desired output
2. Convert the truth-table into a Boolean function
3. Simplify the Boolean function
4. Finally, convert it into a circuit

Gates

Boolean Hardware

Gates

- Electronic devices are made up of *gates*
- Gates take in two inputs and produce a single output
- This is how hardware is used to implement Boolean logic (or any logic)

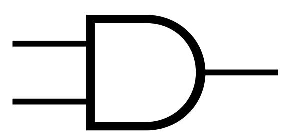
Gates

- Gates can be combined into circuits with *any number of input wires* and a single output wire

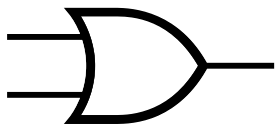
Graphical Representation

- Gates are typically represented using graphical shapes – much like flowcharts
- There are two different competing symbol standards
- We will use the standard, distinct, symbols rather than the IEC (European) ones

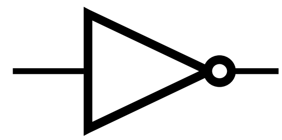
And Gate



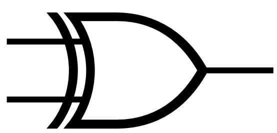
Or Gate



Not Gate



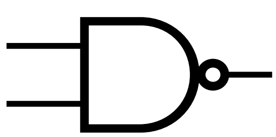
Exclusive Or Gate (aka XOR)



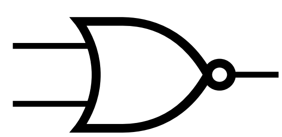
Some Other Gate Symbols

- There are also gate symbols for negated operators
- I won't use these much in class, but it's good to be aware of them (since they are quite common in computer engineering
- For each, note the circle on the output line – it means "not"

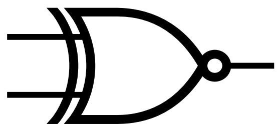
Not And Gate (aka NAND)

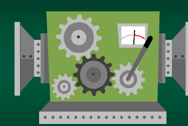


Not Or Gate (aka NOR)



Not Exclusive Or Gate (aka XNOR)



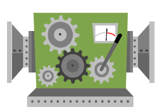


Converting Boolean to Circuits

From Logic to Wires

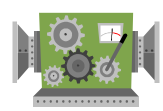
Converting Boolean to Circuits

- Converting from Boolean to circuits maintains a one-to-one correspondence between gates in the circuit and operators in the equation
- But, given an *arbitrary* logic table, how do we realize a circuit for it?



Steps

1. Choose the last operation evaluated
2. Draw a gate and hook up its output
3. Goto 1 until all operations have associated gates
4. Attach the expression inputs



Let's Try One…

- Let's draw a gate representation for the Boolean expression below
- It is actually kinda fun!

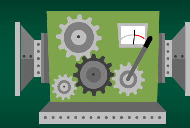
```
(a and b) or ((a or b) and c)
```


Let's Try This…

```
a xor b = (a  and b') or
          (a' and b )
```

Let's Try This…

```
Bidirectional circuit:

(a' + b) * (a + b')
```

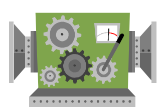



Converting Circuits to Boolean

From Logic to Wires

Converting Circuits to Boolean

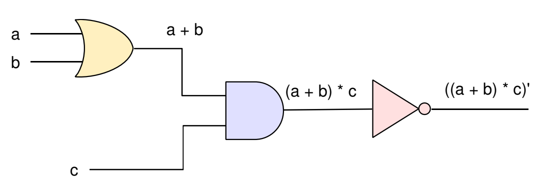
- The other direction is easy too
- Any circuit can be realized as a Boolean expression using the same basic algorithm

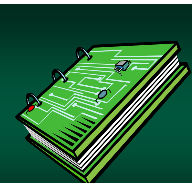


Converting Circuits to Boolean

1. Pick a wire that has a <u>known</u> Boolean value
2. Write *on the wire* a Boolean expression for its value
3. Goto 1 until all wires are complete
4. Circuit's expression written on the circuit's output wire

Example Circuit



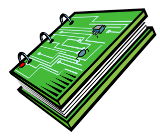


Creating an Arbitrary Circuit

From Truth Table to Wires

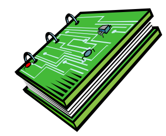
Creating an Arbitrary Circuit

- We converted between Boolean expressions and circuits
- It maintained a one-to-one correspondence between gates in the circuit and operators in the equation

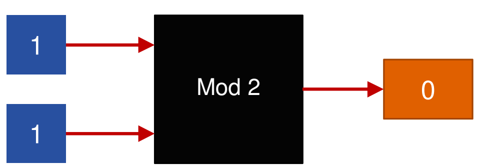


Creating an Arbitrary Circuit

- Given an arbitrary logic table, how do we realize a circuit for it?
- Simple, we look at the inputs that make it true, and write them out in an expression using or's.



Example: 1 Bit Add Mod 2



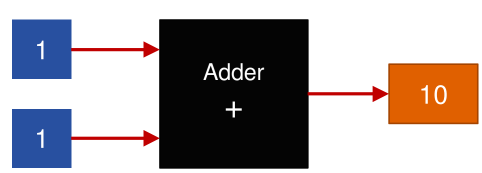
Example: 1 Bit Add Mod 2

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Example: 1 Bit Add Mod 2

```
We want a circuit that is true
when:

(a = F and b = T) or
(a = T and b = F)

out = a' * b + a * b'
```


Example 2: One Bit Adder



Example 2: One Bit Adder

a	b	Out $_1$	Out $_0$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Example: One Bit Adder (Logic)

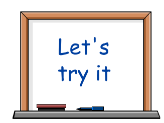
```
out1 = (a = T and b = T)

out0 = (a = F and b = T) or
       (a = T and b = F)
```


Example: One Bit Adder (algebra)

```
out1 = a * b

out0 = a' * b + a * b'
```


Let's Draw the Circuit

- So, we convert the logic of a one-bit adder to logic
- And then to Boolean algebra
- Let's draw how it would be wired on a computer…

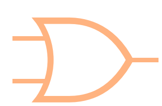


Disjunctive Normal Form

Express Logic With Ease

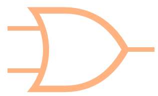
Disjunctive Normal Form

- Best approach to converting tables into circuits is use *Disjunctive Normal Form*
- In this form, the expressions consists of OR's (disjuncts) connecting AND sub-expressions



Definitions

- A *literal* is a Boolean variable v or its complement (e.g. v *or* v')
- A *minterm* of Boolean product $v_1{}^* \, v_2{}^* \ldots v_n$



Definitions

- Hence, a minterm is a "product" of *n* literals, with one literal for each variable
- An equation written only as the "OR" of minterms is in *disjunctive normal form* (also called *sum-of-products* form)

Algorithm

1. Find the rows that indicates a 1 for output (Ignore the ones with 0 as output)
2. Write a minterm for each of them
3. "OR" all the minterms

Example

a	b	y (out)
0	0	1
0	1	1
1	0	0
1	1	0

Example

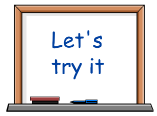
```
DNF of the table is:

y = (a' * b') + (a' * b)

For brevity, for this point on, let's
write as:

y = a'b' + a'b
```

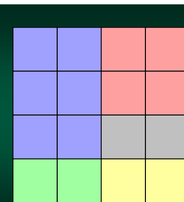

Example

```
We can simply using Boolean algebra:

y = a'b' + a'b
  = a' (b' + b)         Distributive
  = a' (1)              Complement
  = a'                  Identity
```

Let’s Make a 2-Bit Adder

- Let's create the circuit logic for a 2-bit adder
- It will produce a 4-bit result



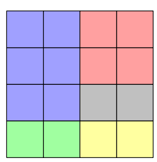


Karnaugh Maps

The Right-Brain Gets to Help

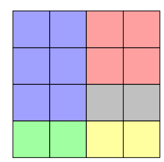
Karnaugh Maps

- A *Karnaugh Map* (pronounced "car-no") is a visual tool to help see relations between minterms.
- A K-Map for *n* variables is a grid of 2^n squares



Karnaugh Maps

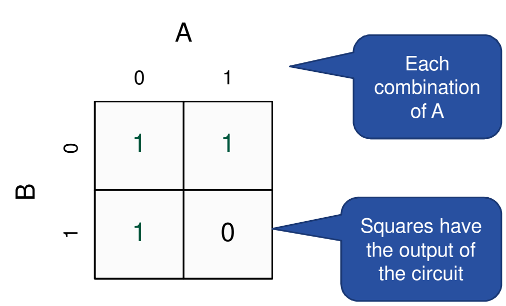
- Every possible minterm of n variables is represented
- It is arranged so that every adjacent pair of squares represent two minterms



Gray Code

- Each square differs in exactly one literal
- This is called *gray code*
  - the values in the table are not ordered in normal ascending order
  - makes it easy to different logical relations
- Important: squares wrap-around to the top and sides

Two-Value K-Map



Three-Variable K-Map

C \ AB	00	01	11	10
0	1	0	1	1
1	0	0	1	1

Four-Variable K-Map

CD \ AB	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

How to Use a K-Map

1. Mark the squares of a K-map corresponding to the function
2. Select a minimal set of rectangles where
   - each rectangle has a power-of-two area and is as large as possible
   - cover every marked square
3. Translate each rectangle into a single midterm and sum all these

How it Works…

- The order of gray code, and the 2^n squares allow us to factor out terminals
- In a rectangle….
  - notice if a terminal changes
  - if so, it factors out to (v + v') which is always true and is meaningless

Example Square: 1×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' B C' D

Example Square: 2×1

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B C' D

Example Square: 1×4

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

C' D

Example Square: 2×2

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

A' D

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

B' D

Example Square: 2×2: Wrapped

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	0	1
10	1	0	0	1

B' D'

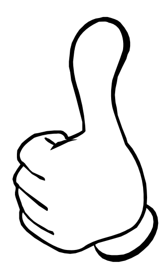
Example Square: 4×2

CD \ AB	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

D

Tips

- There is no magic way to do Step 2. Look and play around until you find the answer
- You can overlap squares – just as long as you "cover" all the 1's



Four-Variable K-Map

CD \ AB	00	01	11	10
00	0	1	1	0
01	1	1	0	0
11	1	1	1	0
10	0	1	1	0

A'D + BC + BC'D'

K-Maps Can Simplify Expressions

- The following is a complex expression that, on the surface, looks difficult to simplify
- K-Maps can help simply expressions.

```
if (a && !b && c  || a && b && !c ||
    a && !b && !c || a && b && c)
```


K-Maps Can Simplify Expressions

- The following is a complex expression that, on the surface, looks difficult to simplify
- K-Maps can help simply expressions.

```
ab'c' + abc' + ab'c + abc
```


K-Maps Can Simplify Expressions

c \ ab	00	01	11	10
0				
1				

ab'c' + abc' + ab'c + abc

K-Maps Can Simplify Expressions

c \ ab	00	01	11	10
0				1
1				

ab'c' + abc' + ab'c + abc

K-Maps Can Simplify Expressions

c \ ab	00	01	11	10
0			1	1
1				

ab'c' + abc' + ab'c + abc

K-Maps Can Simplify Expressions

ab

c	00	01	11	10
0			1	1
1				1

ab'c' + abc' + ab'c + abc

K-Maps Can Simplify Expressions

ab

c	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc

K-Maps Can Simplify Expressions

ab

c	00	01	11	10
0			1	1
1			1	1

ab'c' + abc' + ab'c + abc = a

Efficiency of K-Maps

- A K-Map does not necessarily make the *best* expression/circuit
- All expressions made this way are sums-of-products and some can be made simpler
- e.g. a(b+c) is the same as ab+ac, but uses fewer gate inputs

Don't Care

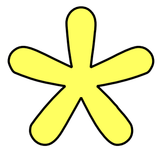
When the Result is Meaningless

Don't Care

- Sometimes *we don't really care* what output the circuit generates for some combinations of inputs
- So, for those inputs, the results are simply not significant

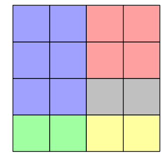
Don't Care

- In truth tables, the value "Don't Care" is represented with an asterisk
- It can be considered True or False – whichever is more *convenient* for the circuit



Karnaugh Maps and Don't Care

- We can construct a Karnaugh Map like before
- Except the squares corresponding to don't care outputs are marked (with an asterisk)



Karnaugh Maps and Don't Care

- Then, when outlining blocks, we can (at our convenience) consider the "don't care" squares as either 0 or 1
- Since we want to make the largest outlines possible, we will sometimes consider a don't care to be true, and sometimes false

Example

We want to guarantee that the output of a circuit is 1 if both inputs are 1

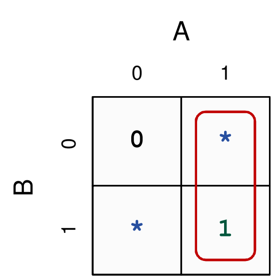
And 0 when both inputs are 0

But otherwise we do not care

Example

x	y	out
0	0	0
0	1	*
1	0	*
1	1	1

K-Map For The Example



We don't need B!

out = A

… or we can do this

	A=0	A=1
B=0	0	*
B=1	*	1

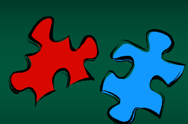
Just B!

out = B

Four-Variable (with Don't Care)

CD \ AB	00	01	11	10
00	0	1	1	*
01	1	1	*	0
11	1	1	1	0
10	*	1	1	0

out = B + A'D

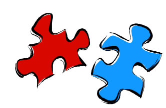


Functional Completeness

Just How Much Do We Need?

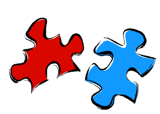
Functional Completeness

- We can construct a circuit for any Boolean expression using and / or / not
- This means the set of gates {and, or, not} is *functionally complete*



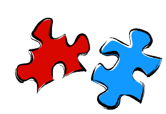
Function Completeness

- However, we don't need all three gates
- DeMorgan's laws shows us that we can construct:
  - an OR using an AND
  - and AND using an OR



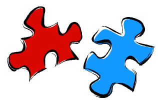
We Don't Need Or!

- So {and, not} are also complete because by DeMorgan's Law:
  $x + y = (x'y')'$
- So, any expression that can be written using {and, or, not} can be written using just {and, not}



or… We Don't Need And!

- Also {or, not} is functionally complete since xy = (x'+y') '
- So, any expression that can be written using {and, or, not} can be written using just {or, not}



Functional Completeness

- So, are any of the singular sets {and}, {or}, {not} functionally complete?
- In other words, can and/or/not all be converted into a single type of gate?
- **No.** Neither {and} or {or} can be converted to a {not}

NAND

- So, is there a gate that can, alone, be functional complete?
- What about NAND (negated And)?
  - x nand y = (xy)'
  - Note: the NAND gate is not implemented with an AND gate and a NOT gate. It just has the same truth table as (xy)'

NAND

- To show that {nand} is functionally complete, we need to show that we can implement {and, or, not} using it
- The result would be greatly beneficial!
  - we would have to just construct 1 gate to create any circuit
  - this would greatly aid construction

Not → Nand

```
Converting not to nand:

x'  =  x'
    =  (xx)'        Idempotent
    =  x nand x     nand format
```

We can implement NOT by using a NAND. Both input will be x

Or → Nand

```
Note: x' = x nand x

x + y =  x + y
      =  (x'y')'                  DeMorgan
      =  x' nand y'               nand format
      =  (x nand x) nand (y nand y)
```

Last proof let us convert NOT into NAND

And → Nand

```
Note: x' = x nand x

xy     =  xy
       =  (x nand y)'       Negate nand
       =  (x nand y) nand (x nand y)
```

Last proof let us convert NOT into NAND

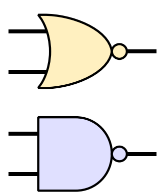
Summary

- The expressions below show that nand can be used to implement NOT, OR, AND
- So, we can just use NAND since it is *functionally complete*

```
x'     =  x nand x
xy     =  (x nand y) nand (x nand y)
x + y  =  (x nand x) nand (y nand y)
```


How Hardware Works

- Also NOR is functionally complete
- P NOR Q = (P + Q)'
- Hardware can alternatively use this gate rather than NAND



How Hardware Works

- If our hardware can just implement NAND or NOR, then we can create a circuit with just one gate
- In fact, many fabrication processes use only NAND or NOR gates