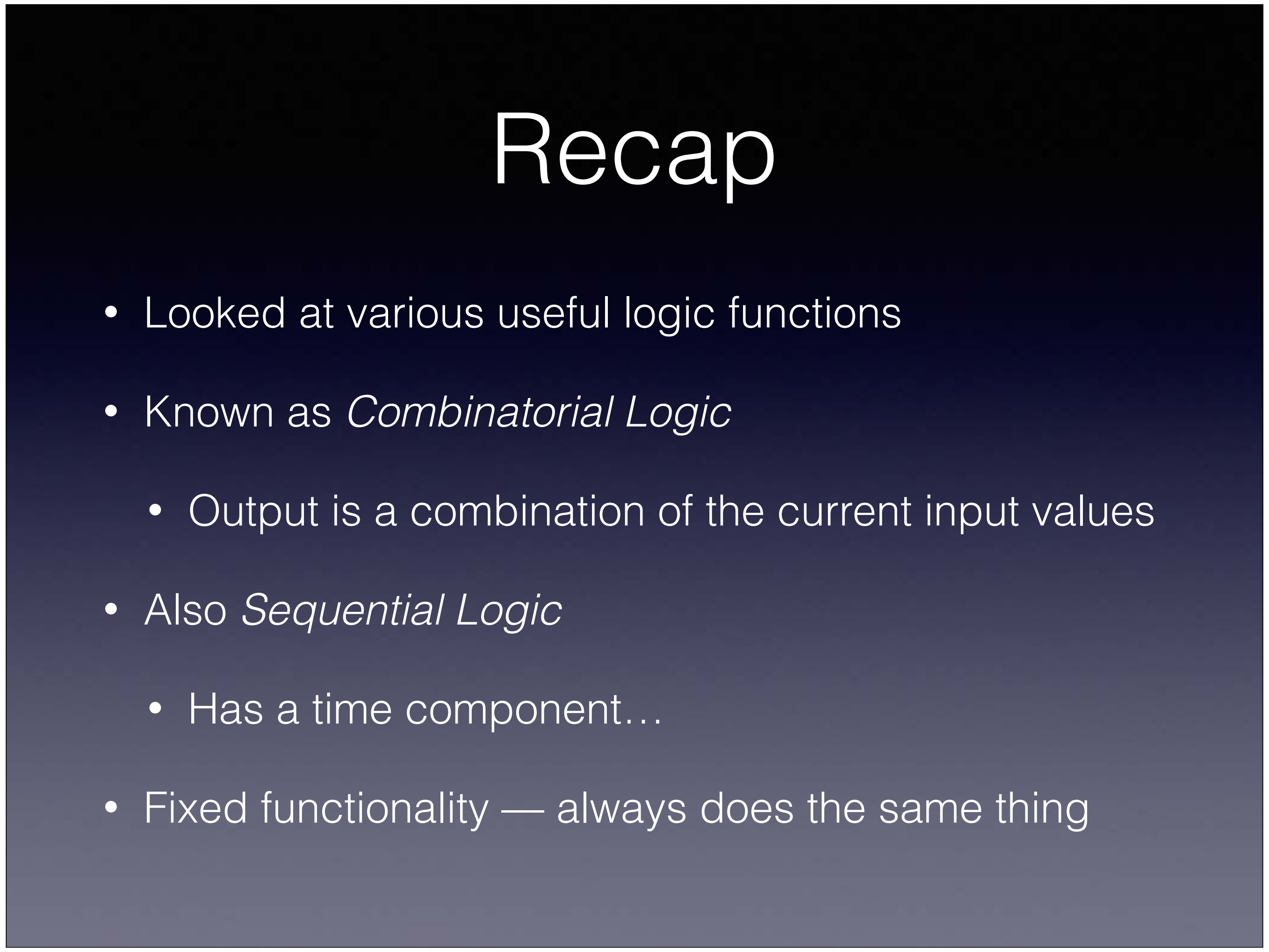
# ALU

Steven R. Bagley



## Recap

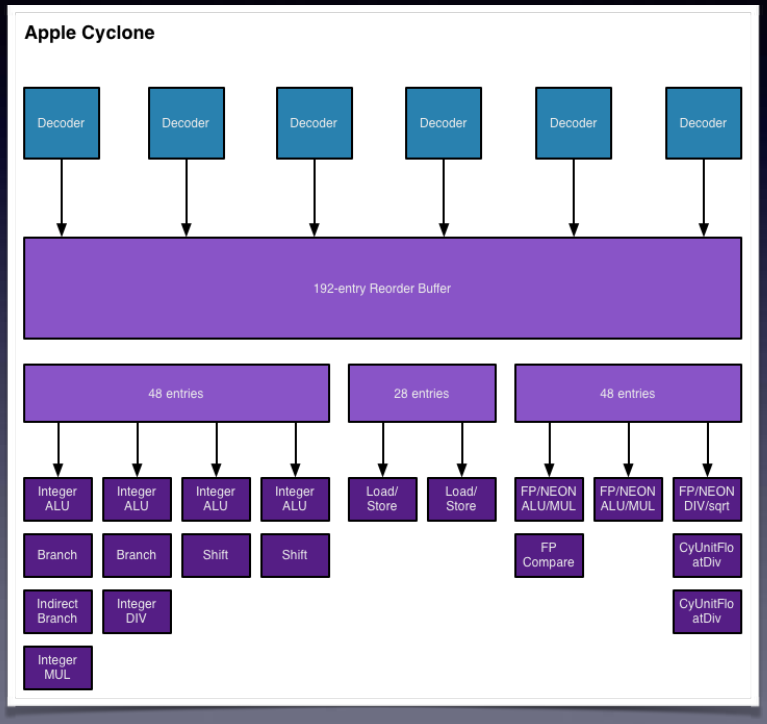
- Looked at various useful logic functions
- Known as *Combinatorial Logic*
  - Output is a combination of the current input values
- Also *Sequential Logic*
  - Has a time component…
- Fixed functionality — always does the same thing

Mention propagation delay

## ALU

- Arithmetic and Logic Unit
- Heart of the CPU performs all the arithmetic and logic operations of the CPU
- Combinatorial Logic
- All CPUs have one, some have several

# Apple A7



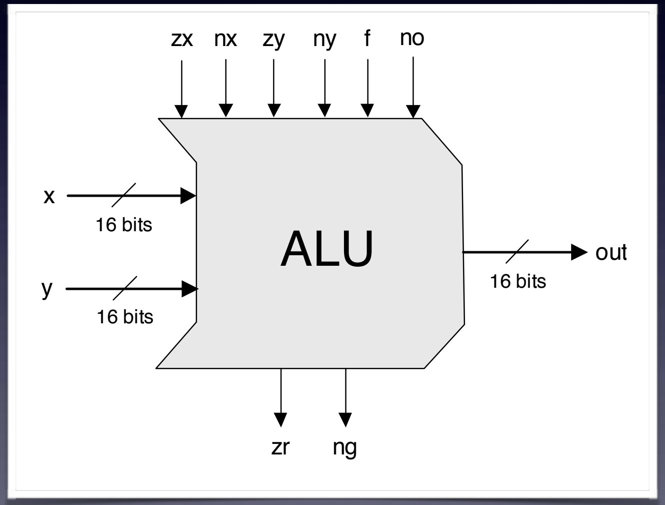
See http://www.anandtech.com/show/7910/apples-cyclone-microarchitecture-detailed

By having four ALUs the CPU can do four operations at once, providing they don’t rely on the answer from each other

## ALU

- Usually considered to have two inputs
- And one output
- These are usually multi-bit
- Use 8-bits in our example…
- Takes the two inputs, performs an operation and produces an output

# ALU

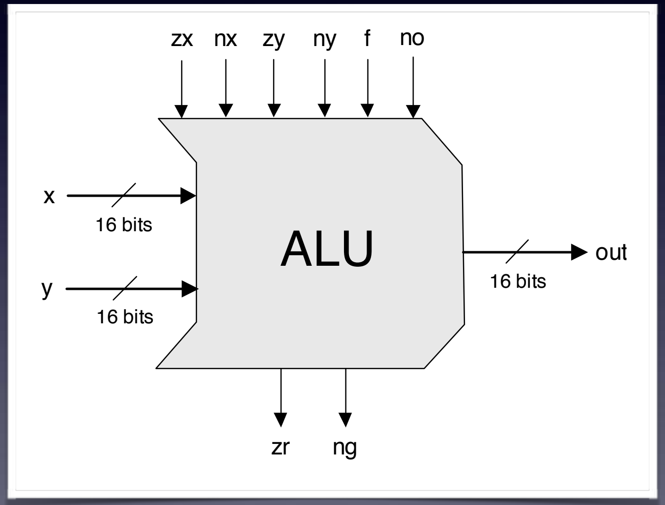


What sort of operations do you want it to do?
Think about the operators we've had in G51PRG

## ALU functionality

- ALU functionality is *dynamic* — it can change
    - First, add then later subtract
- Need a mechanism to select which function the ALU performs
- Done by having a set of control pins that *select* what function the ALU is to perform

# ALU



Now with added function selection pins

## ALU functionality

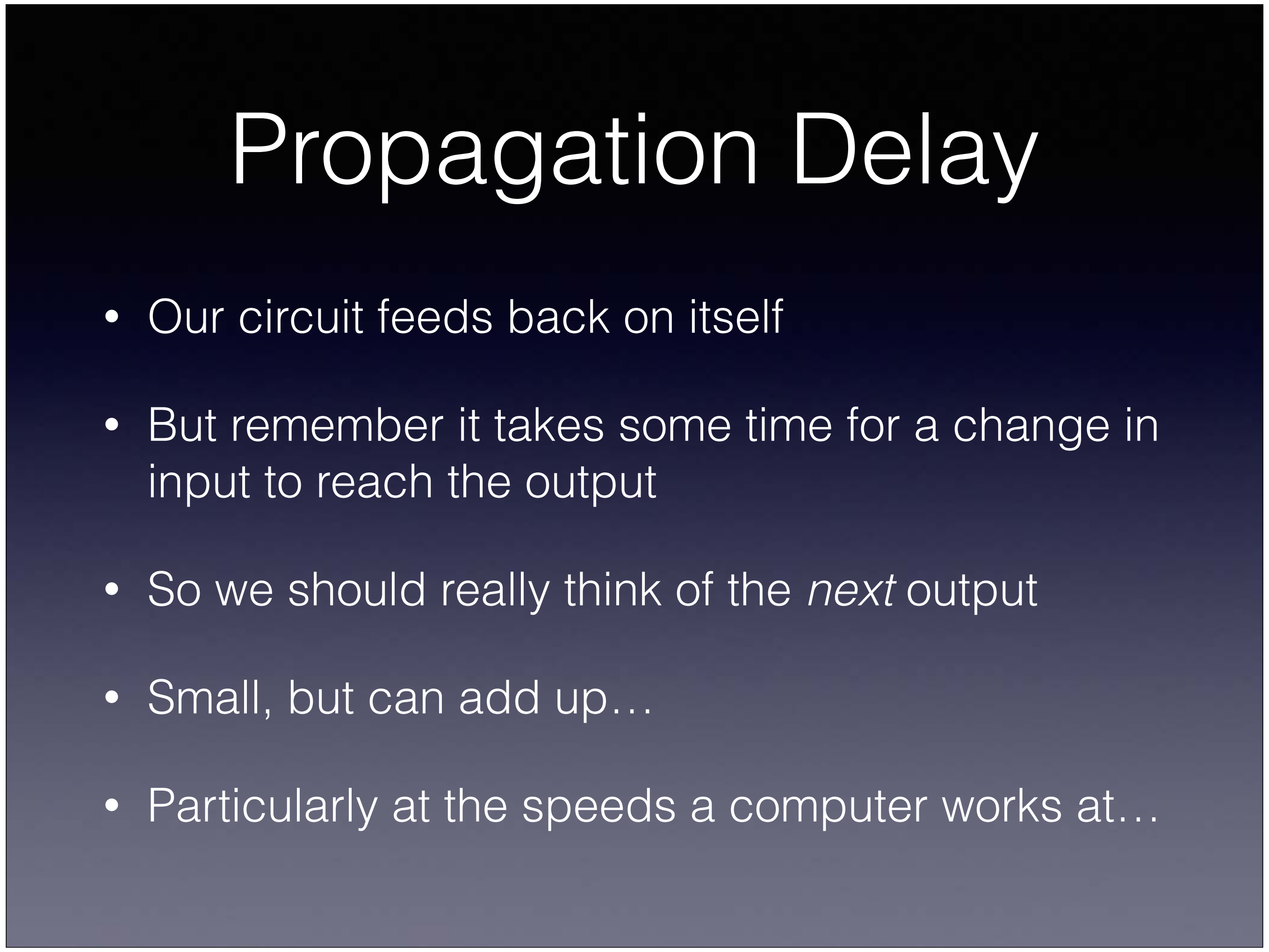
- Lots of possible ALU functions
  - Addition, subtraction, AND/OR/NOT, setting values
- However, we've already seen how some of these operations can be produced in terms of others…
- Want the minimum amount of logic that can perform these tasks…

## ALU functionality

- Lots of possible ALU functions
    - Addition, ~~subtraction~~, AND/~~OR~~/NOT, setting values
- However, we've already seen how some of these operations can be produced in terms of others…
- Want the minimum amount of logic that can perform these tasks…

## ALU functionality

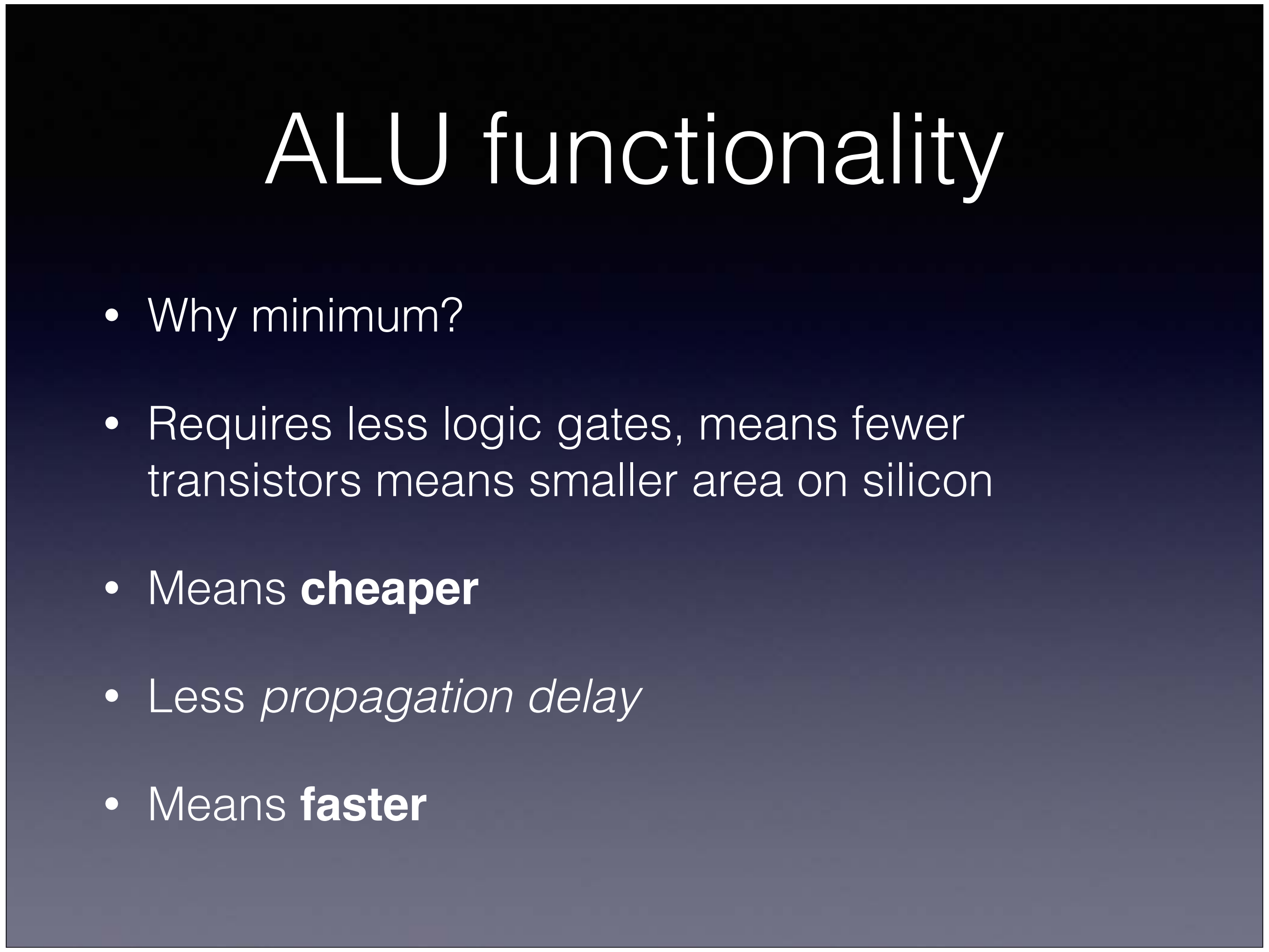
- Why minimum?
- Requires less logic gates, means fewer transistors means smaller area on silicon
- Means **cheaper**
- Less *propagation delay*



## Propagation Delay

- Our circuit feeds back on itself
- But remember it takes some time for a change in input to reach the output
- So we should really think of the *next* output
- Small, but can add up…
- Particularly at the speeds a computer works at…

Around 5-12ns

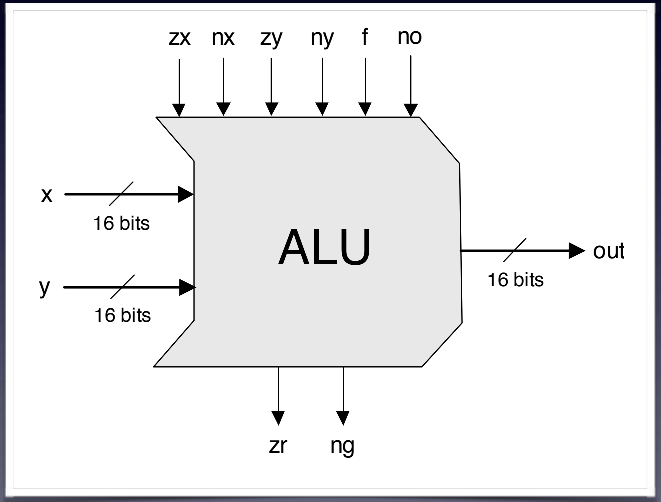


## ALU functionality

- Why minimum?
- Requires less logic gates, means fewer transistors means smaller area on silicon
- Means **cheaper**
- Less *propagation delay*
- Means **faster**

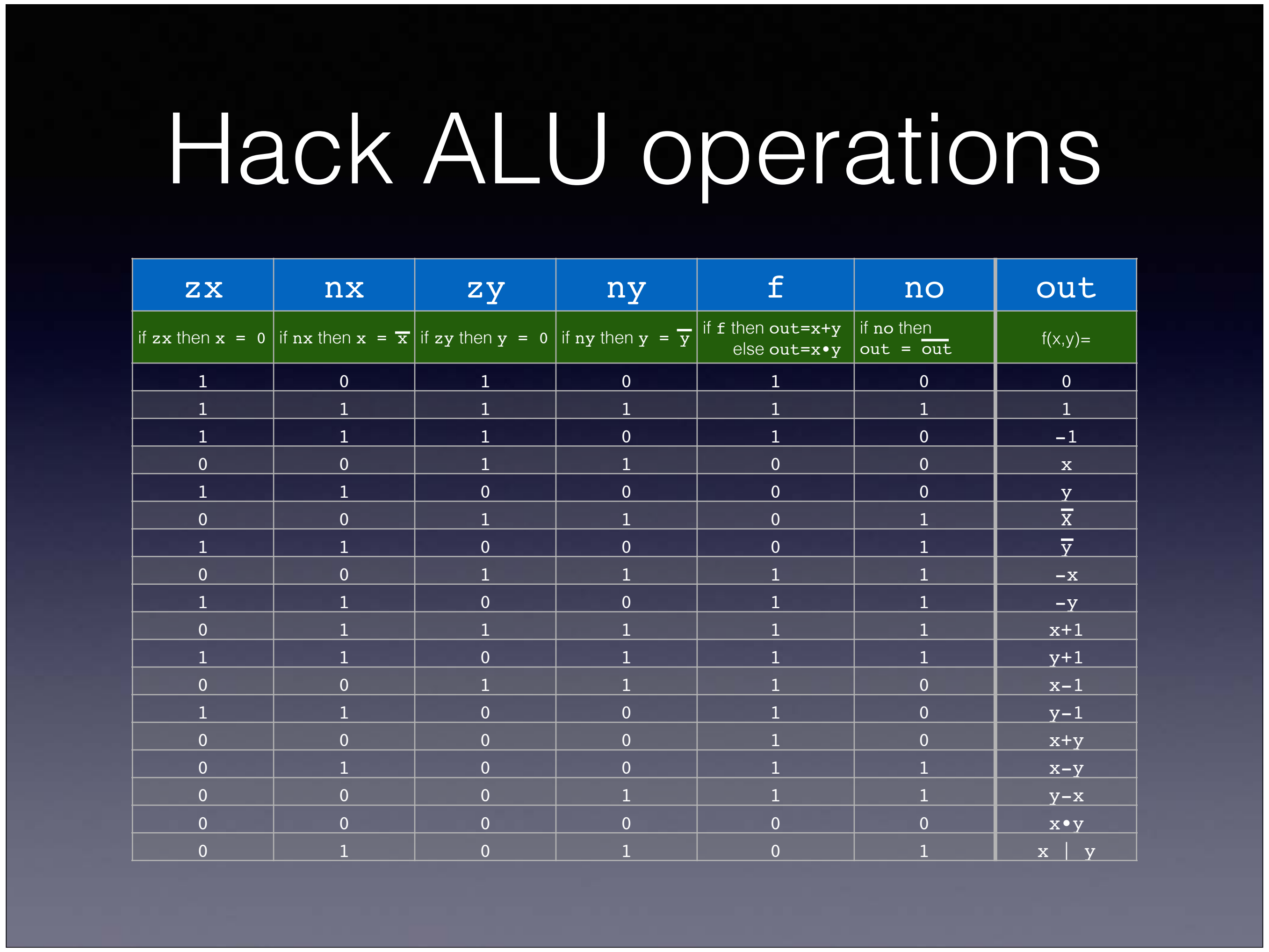
Worth doing the maths/logic to work out the better circuit :)
Let’s look at a sample ALU…

## Hack ALU



| Input | Meaning |
|---|---|
| zx | if zx then x = 0 |
| nx | if nx then x = $\overline{x}$ |
| zy | if zy then y = 0 |
| ny | if ny then y = $\overline{y}$ |
| f | if f then out=x+y<br>else out=x•y |
| no | if no then out = $\overline{out}$ |

From nand2tetris, Stages are sequential in the order shown so X can be zeroed, and inverted…
The output is added or anded then inverted (not)
Explain bitwise and
Show them an equivalent C function for this bit of hardware…

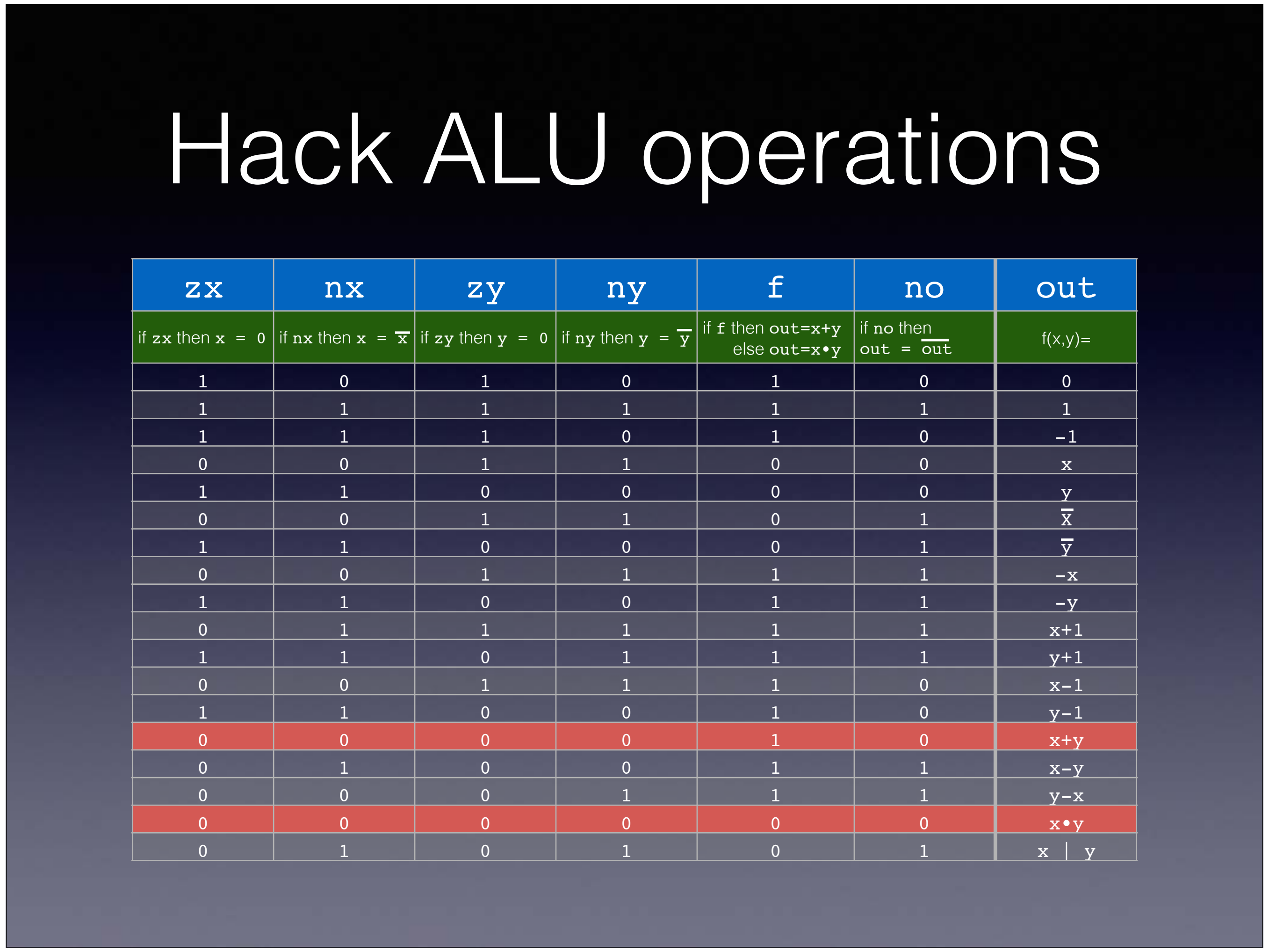


# Hack ALU operations

| zx | nx | zy | ny | f | no | out |
|---|---|---|---|---|---|---|
| if zx then x = 0 | if nx then x = $\overline{x}$ | if zy then y = 0 | if ny then y = $\overline{y}$ | if f then out=x+y else out=x•y | if no then out = $\overline{\text{out}}$ | f(x,y)= |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | -1 |
| 0 | 0 | 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 0 | 0 | 0 | y |
| 0 | 0 | 1 | 1 | 0 | 1 | $\overline{x}$ |
| 1 | 1 | 0 | 0 | 0 | 1 | $\overline{y}$ |
| 0 | 0 | 1 | 1 | 1 | 1 | -x |
| 1 | 1 | 0 | 0 | 1 | 1 | -y |
| 0 | 1 | 1 | 1 | 1 | 1 | x+1 |
| 1 | 1 | 0 | 1 | 1 | 1 | y+1 |
| 0 | 0 | 1 | 1 | 1 | 0 | x-1 |
| 1 | 1 | 0 | 0 | 1 | 0 | y-1 |
| 0 | 0 | 0 | 0 | 1 | 0 | x+y |
| 0 | 1 | 0 | 0 | 1 | 1 | x-y |
| 0 | 0 | 0 | 1 | 1 | 1 | y-x |
| 0 | 0 | 0 | 0 | 0 | 0 | x•y |
| 0 | 1 | 0 | 1 | 0 | 1 | x \| y |

From nand2tetris, Stages are sequential in the order shown so X can be zeroed, and inverted…
The output is added or anded then inverted (not)

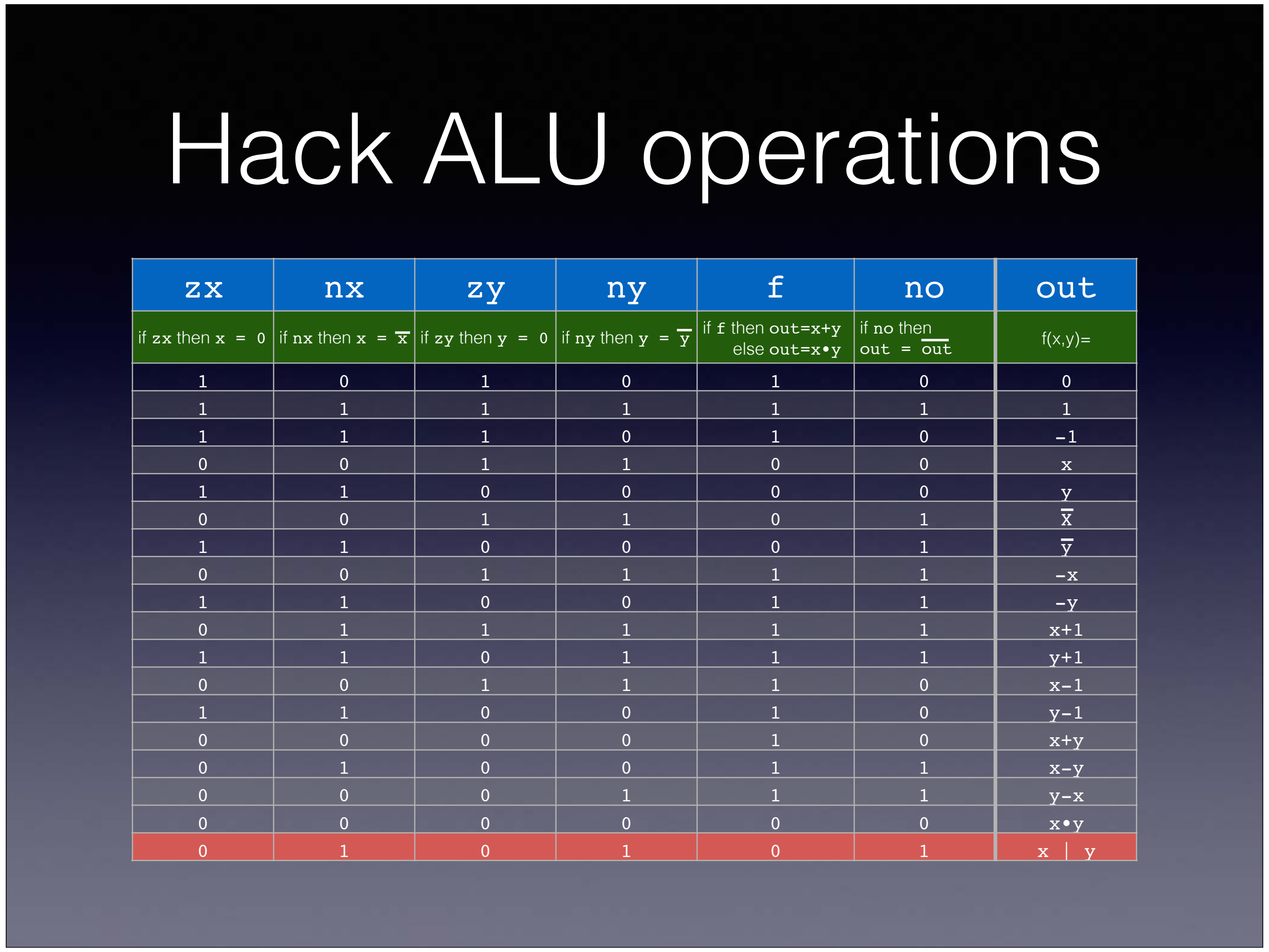
Note: I've used the | as or to distinguish it from the numerical add (with +)



## Hack ALU operations

| zx | nx | zy | ny | f | no | out |
|---|---|---|---|---|---|---|
| if zx then x = 0 | if nx then x = $\bar{x}$ | if zy then y = 0 | if ny then y = $\bar{y}$ | if f then out=x+y else out=x•y | if no then out = $\overline{\text{out}}$ | f(x,y)= |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | -1 |
| 0 | 0 | 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 0 | 0 | 0 | y |
| 0 | 0 | 1 | 1 | 0 | 1 | $\bar{x}$ |
| 1 | 1 | 0 | 0 | 0 | 1 | $\bar{y}$ |
| 0 | 0 | 1 | 1 | 1 | 1 | -x |
| 1 | 1 | 0 | 0 | 1 | 1 | -y |
| 0 | 1 | 1 | 1 | 1 | 1 | x+1 |
| 1 | 1 | 0 | 1 | 1 | 1 | y+1 |
| 0 | 0 | 1 | 1 | 1 | 0 | x-1 |
| 1 | 1 | 0 | 0 | 1 | 0 | y-1 |
| 0 | 0 | 0 | 0 | 1 | 0 | x+y |
| 0 | 1 | 0 | 0 | 1 | 1 | x-y |
| 0 | 0 | 0 | 1 | 1 | 1 | y-x |
| 0 | 0 | 0 | 0 | 0 | 0 | x•y |
| 0 | 1 | 0 | 1 | 0 | 1 | x \| y |

Fairly obvious what is happening with these here — the f control input is being used to decided between adding and ‘anding’ the input here



# Hack ALU operations

| zx | nx | zy | ny | f | no | out |
|---|---|---|---|---|---|---|
| if zx then x = 0 | if nx then x = $\overline{x}$ | if zy then y = 0 | if ny then y = $\overline{y}$ | if f then out=x+y else out=x•y | if no then out = $\overline{\text{out}}$ | f(x,y)= |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | -1 |
| 0 | 0 | 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 0 | 0 | 0 | y |
| 0 | 0 | 1 | 1 | 0 | 1 | $\overline{x}$ |
| 1 | 1 | 0 | 0 | 0 | 1 | $\overline{y}$ |
| 0 | 0 | 1 | 1 | 1 | 1 | -x |
| 1 | 1 | 0 | 0 | 1 | 1 | -y |
| 0 | 1 | 1 | 1 | 1 | 1 | x+1 |
| 1 | 1 | 0 | 1 | 1 | 1 | y+1 |
| 0 | 0 | 1 | 1 | 1 | 0 | x-1 |
| 1 | 1 | 0 | 0 | 1 | 0 | y-1 |
| 0 | 0 | 0 | 0 | 1 | 0 | x+y |
| 0 | 1 | 0 | 0 | 1 | 1 | x-y |
| 0 | 0 | 0 | 1 | 1 | 1 | y-x |
| 0 | 0 | 0 | 0 | 0 | 0 | x•y |
| 0 | 1 | 0 | 1 | 0 | 1 | $x \mid y$ |

Relatively obvious — using De Morgan’s law to produce an OR of the two inputs
Some of the others are less than straightforward (starting with the second one)

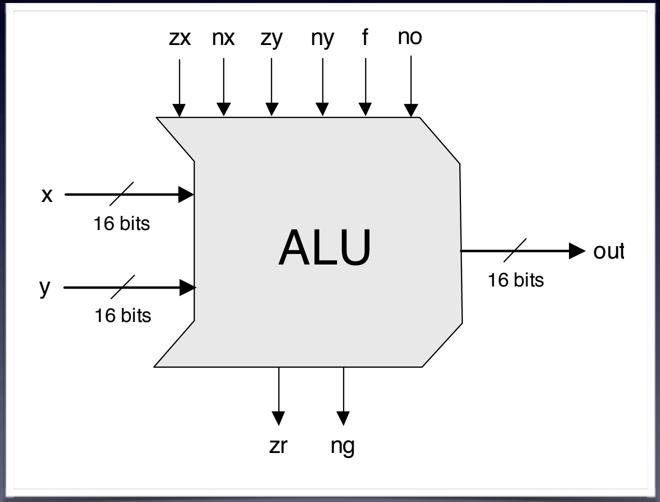
## Hack ALU functionality

- From a few relatively simple arrangement of logic gates
- It is possible to create an ALU that can perform many operations
- Not necessarily straightforward to understand how they work — some rely on a blend of mathematical and logical operations to produce the result

## Designing ALU

- Start with the functions you want it to do
- Then think about what arrangements of switchable logic functions can provide those functions

# Hack ALU

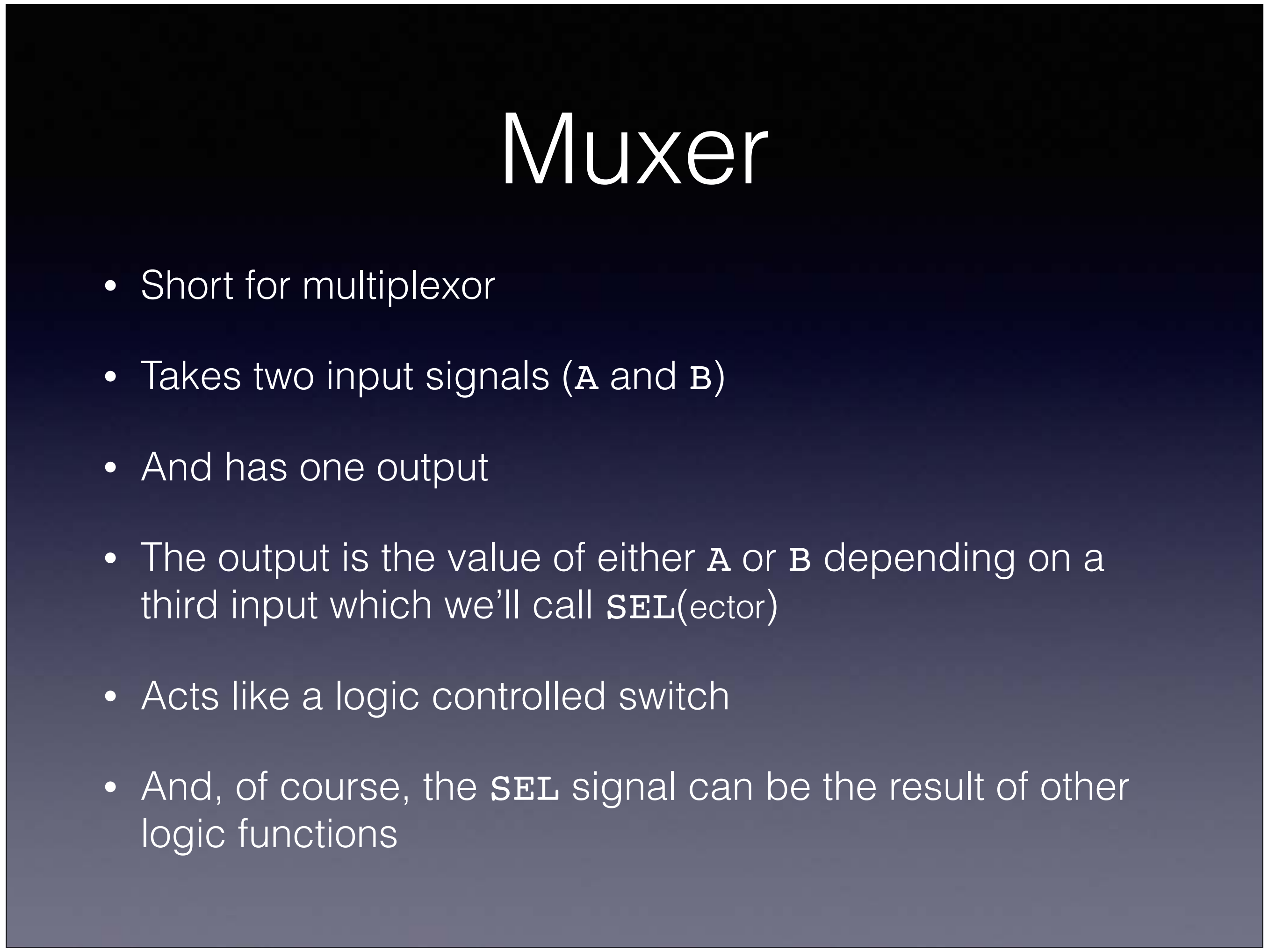


| Input | Meaning |
|---|---|
| `zx` | if `zx` then `x = 0` |
| `nx` | if `nx` then `x` = $\overline{\texttt{x}}$ |
| `zy` | if `zy` then `y = 0` |
| `ny` | if `ny` then `y` = $\overline{\texttt{y}}$ |
| `f` | if `f` then `out=x+y` else `out=x•y` |
| `no` | if `no` then `out` = $\overline{\texttt{out}}$ |

From nand2tetris, Stages are sequential in the order shown so X can be zeroed, and inverted…
The output is added or anded then inverted (not)
Explain bitwise and
Show them an equivalent C function for this bit of hardware…

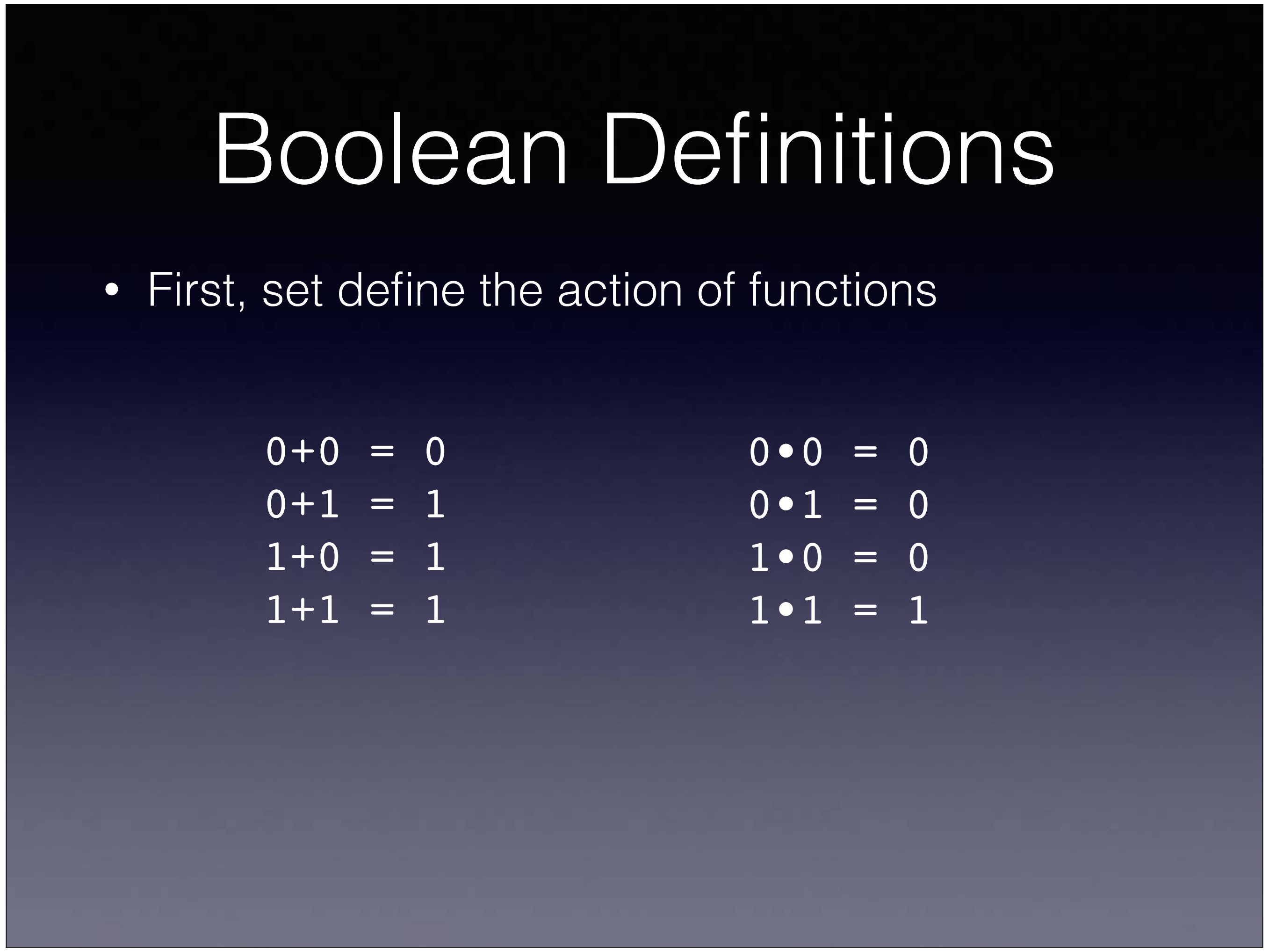
## Logical Switching

- ALU needs to be able to switch between two sets of values
  - e.g. between **`x`** and **`0`**
  - Or between **`out`** and inverted **`out`**
- Need a new logic function to handle this, the *muxer*


## Muxer

- Short for multiplexor
- Takes two input signals (`A` and `B`)
- And has one output
- The output is the value of either `A` or `B` depending on a third input which we'll call `SEL`(ector)
- Acts like a logic controlled switch
- And, of course, the `SEL` signal can be the result of other logic functions

Show how to create this in the standard way
Then show how to simplify it down into purely NAND gates

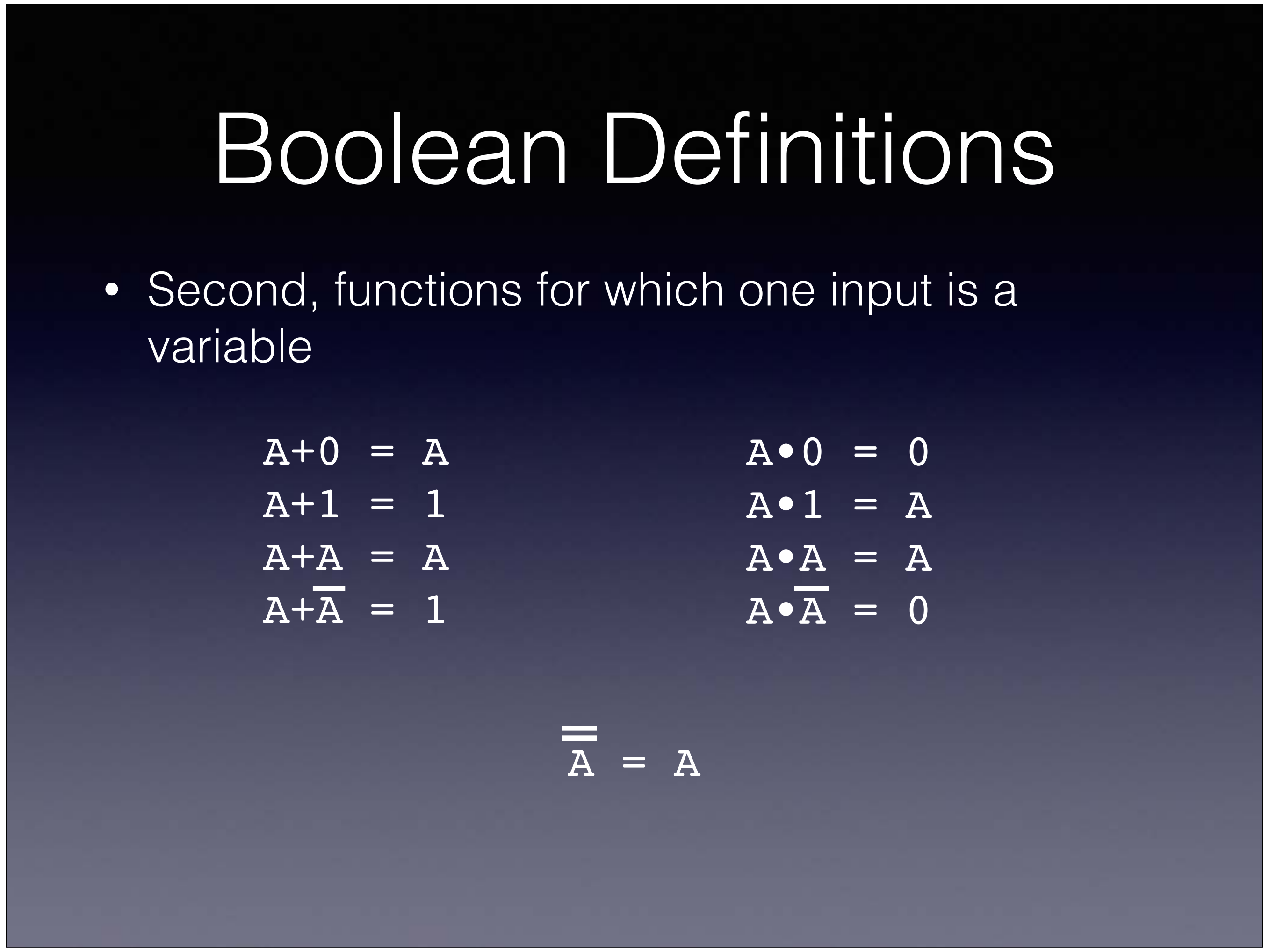


## Boolean Definitions

- First, set define the action of functions

```
0+0  =  0          0•0  =  0
0+1  =  1          0•1  =  0
1+0  =  1          1•0  =  0
1+1  =  1          1•1  =  1
```

Equivalent to the truth table



## Boolean Definitions

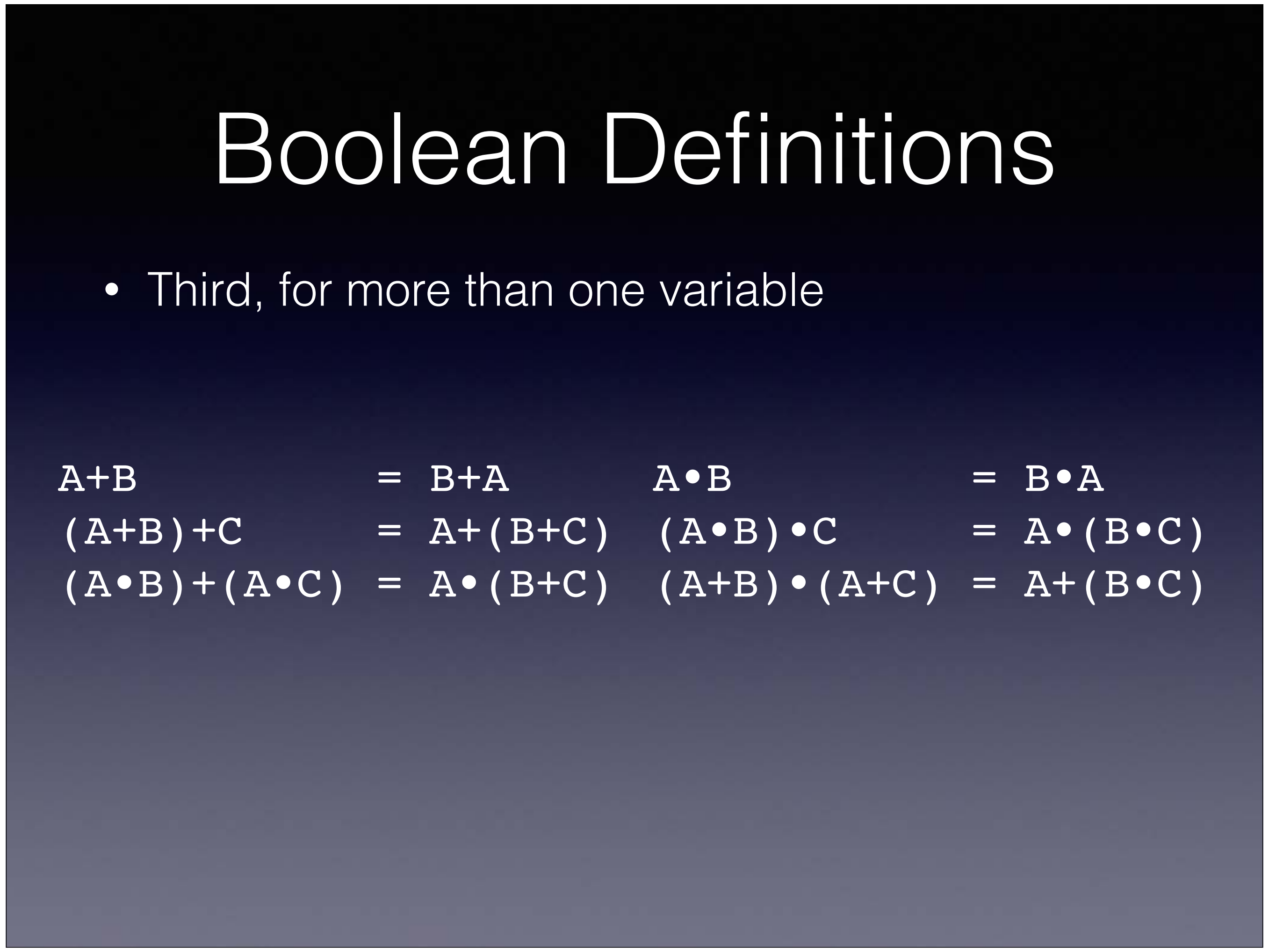
- Second, functions for which one input is a variable

| | |
|---|---|
| $A+0 = A$ | $A \bullet 0 = 0$ |
| $A+1 = 1$ | $A \bullet 1 = A$ |
| $A+A = A$ | $A \bullet A = A$ |
| $A+\bar{A} = 1$ | $A \bullet \bar{A} = 0$ |

$$\bar{\bar{A}} = A$$

Go through them

Last one is NOT of NOT A

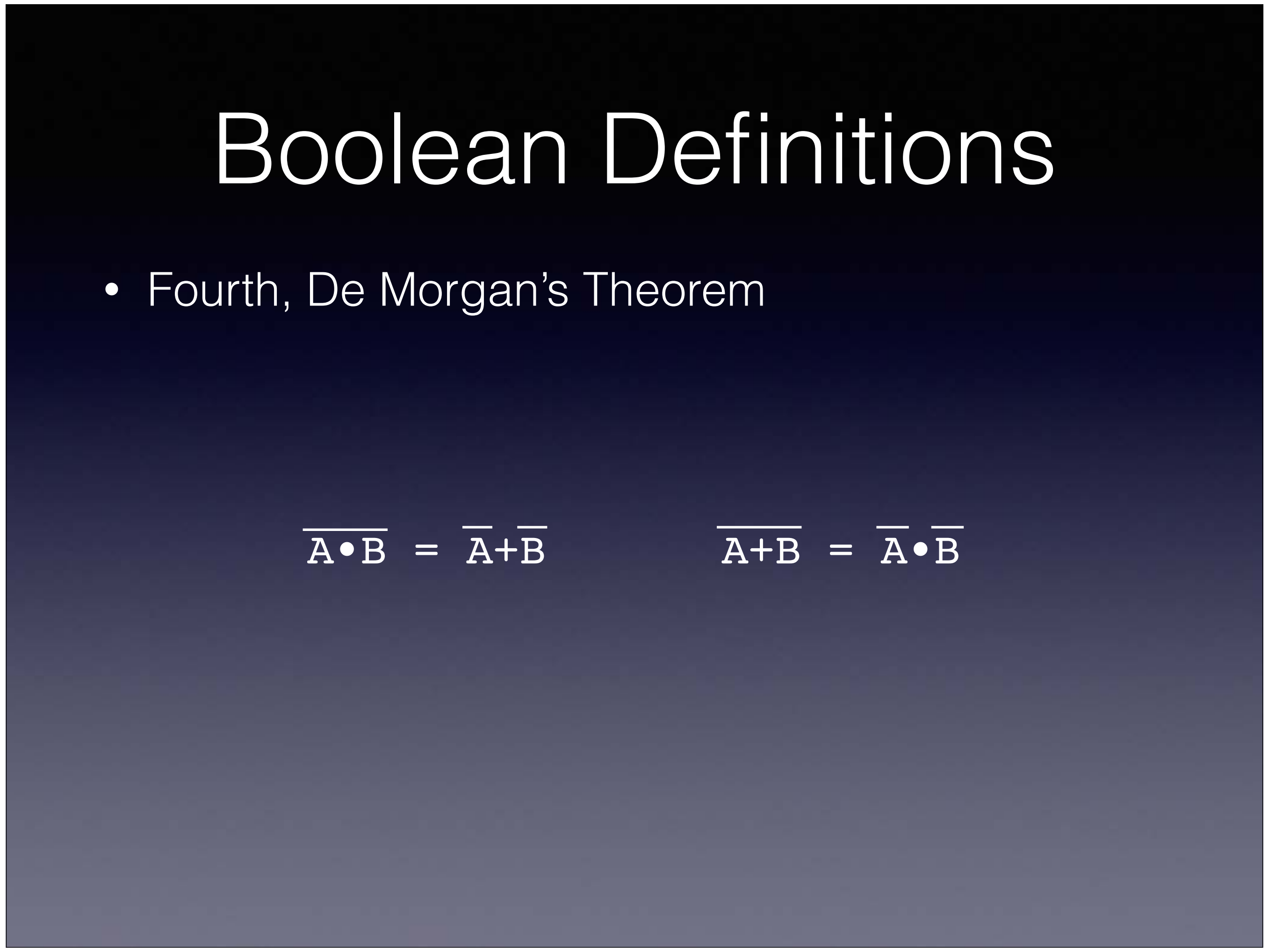


## Boolean Definitions

- Third, for more than one variable

$$\begin{array}{llll}
A+B & = B+A & A\bullet B & = B\bullet A \\
(A+B)+C & = A+(B+C) & (A\bullet B)\bullet C & = A\bullet(B\bullet C) \\
(A\bullet B)+(A\bullet C) & = A\bullet(B+C) & (A+B)\bullet(A+C) & = A+(B\bullet C)
\end{array}$$

Go through them



## Boolean Definitions

- Fourth, De Morgan's Theorem

$\overline{A \bullet B} = \overline{A} + \overline{B}$ $\overline{A+B} = \overline{A} \bullet \overline{B}$

Go through them
These rules are what lets us build everything out of NAND gates
Show truth tables on board for these