

# CS3102 Theory of Computation

4 [www.cs.virginia.edu/~njb2b/cstheory/s2020](http://www.cs.virginia.edu/~njb2b/cstheory/s2020)

Warm up:

Software can be reconfigured, hardware cannot.

When we build hardware, how do we decide what to implement?

If you found the fitbit, the owner is down front, and is upset about all her wasted steps from the past 2 days.

1) Specialized -  
what functions are important  
for graphics?  
• math.

★ • General - it does what you  
need software needs, it too

# Logistics

- Exercise 2 due Tuesday
- Quiz 3 released Friday
- Exercise 3 is out this weekend
  - Last “regular-sized” exercise before midterm
  - There will be a “tiny” exercise 4 due the Tuesday before the midterm

# Last Time

- Boolean Circuits as a model of computing
- Straightline Programs as a model of computing
- Proved  $\text{NAND-Straightline} = \text{NAND-Circ} = \text{AON-Circ} = \text{AON-straightline}$

# Lookup

- Indexing into a bitstring
- The *Lookup* function of order  $k$ :

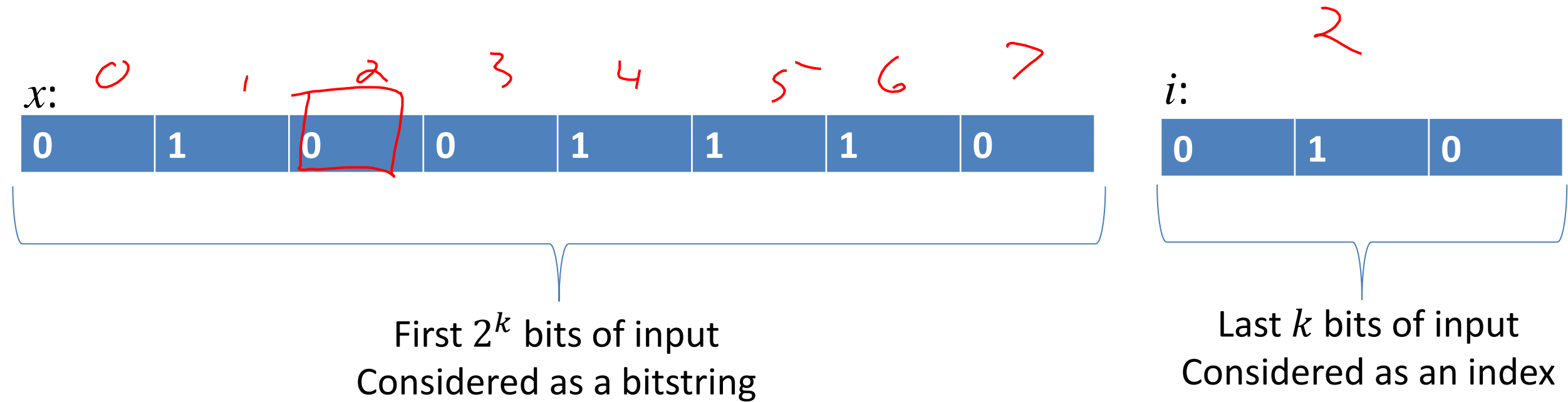
$$LOOKUP_k: \{0,1\}^{2^k + k} \rightarrow \{0,1\}$$

Defined such that for  $x \in \{0,1\}^{2^k}$ ,  $i \in \{0,1\}^k$ :

$$LOOKUP_k(\underline{x}, \underline{i}) = \underline{x_i}$$

# $LOOKUP_k$

$k = 3$



# Theorem

There is a NAND-Circuit that computes

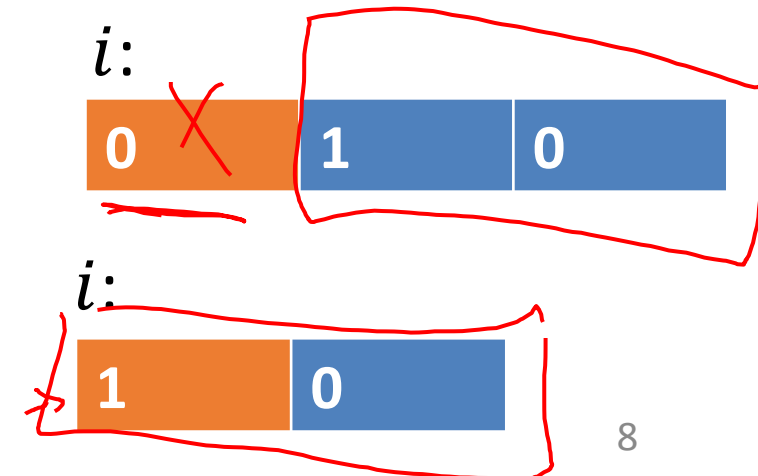
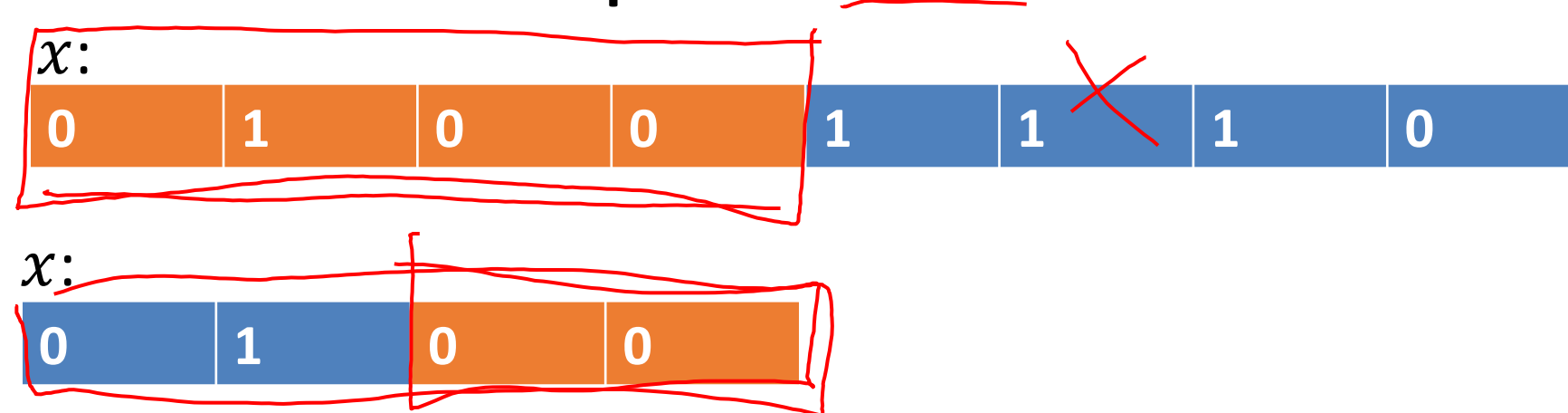
$$LOOKUP_k: \{0,1\}^{2^k+k} \rightarrow \{0,1\}$$

Moreover, the number of gates required is at

$$\text{most } \underline{4 \cdot 2^k - 4}$$

# Proof idea

- Consider index  $i$
- If the first bit of  $i$  is 0, then the bit we're looking for is in the first half of  $x$
- Do lookup for  $k - 1$






# Defining $LOOKUP_k$

For  $k \geq 2$ ,  $LOOKUP_k(x_0, \dots, x_{2^k-1}, i_0, \dots, i_{k-1})$  is equal to:

$$IF(i_0, LOOKUP_{k-1}(x_{2^{k-1}}, \dots, x_{2^k-1}, i_1, \dots, i_{k-1}), LOOKUP_{k-1}(x_0, \dots, x_{2^{k-1}-1}, i_1, \dots, i_{k-1}))$$

# Base Case

```
def LOOKUP1(x0, x1, i0):  
    return IF(i0, x1, x0)
```

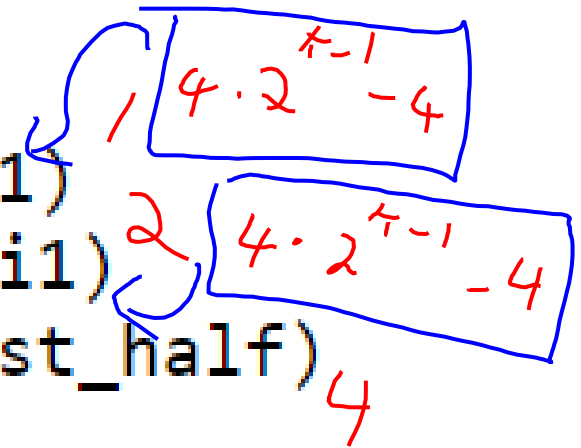


# Next Step

# LOOKUP2

$$\underline{2 \left( 4 \cdot 2^{t-1} - 4 \right) + 4}$$

```
def LOOKUP2(x0,x1,x2,x3,i0,i1):  
    first_half = LOOKUP1(x0,x1,i1)  
    second_half = LOOKUP1(x2,x3,i1)  
    return IF(i0,second_half,first_half)
```



LOOKUP3 and 4

# Counting Gates

Show this uses at most  $4 \cdot 2^k - 4$  gates (lines of code)

Base case:  $\pi = 1$

$\rightarrow \text{lookup}_\pi$  uses  $4 \text{ gates} \leq \underline{4 \cdot 2^1 - 4 = 4}$

hyp: for  $\pi-1$ ,  $\text{lookup}_{\pi-1}$  uses  $\leq 4 \cdot 2^{\pi-1} - 4$  gates

# Counting Gates

Show this uses at most  $4 \cdot 2^k - 4$  gates (lines of code)

$100\pi \cup P_{\pi}$  uses  $2 \cdot (\# \text{ gates } + \text{max } 100\pi \cup P_{\pi-1} \text{ needs}) + 4$

$$2(4 \cdot 2^{\pi-1} - 4) + 4 = 4 \cdot 2^{\pi} - 8 + 4 = 4 \cdot 2^{\pi} - 4 \quad \checkmark$$

$4 \cdot 2^{\pi}$

# Computing Every Finite Function


- Next we'll show that NAND is "universal"
- Any finite function can be computed by some NAND-straightline program (equivalently, a NAND-circuit)

AON-Circ

AON-SL

# Idea

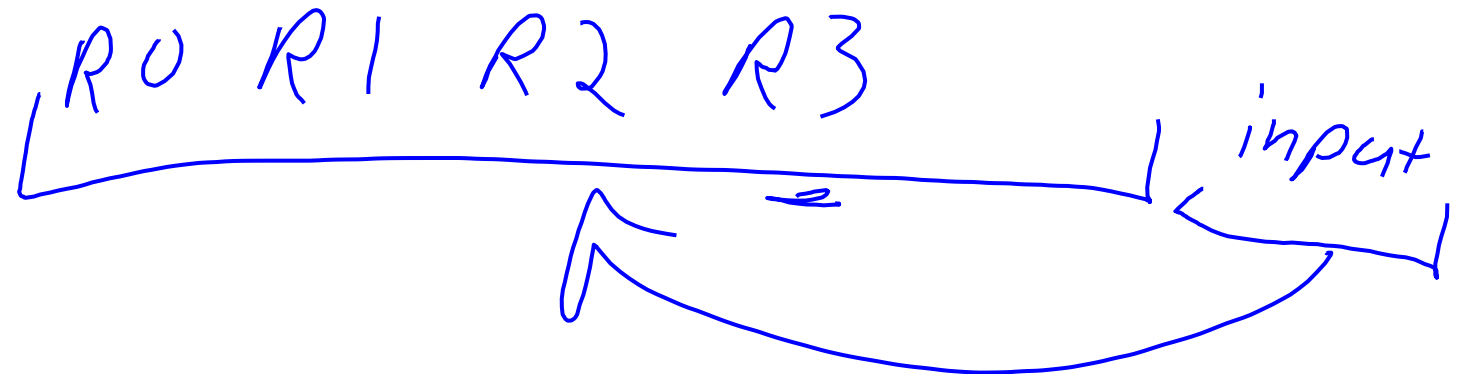
Consider the function  $f: \{0,1\}^3 \rightarrow \{0,1\}$



Handwritten labels R0, R1, R2, R3 are placed to the left of the table rows.

Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

We will have one variable to represent each possible input. We'll do a lookup with the actual input to select the proper output



# Straightline Code for F

```
def F(x0,x1,x2):
```

```
    F000=0
```

```
    F001=0
```

```
    F010=1
```

```
    F011=0
```

```
    F100=1
```

```
    F101=1
```

```
    F110=0
```

```
    F111=0
```

```
    return LOOKUP3(F000,F001,F010,F011,F100,F101,F110,F111,x0,x1,x2)
```

Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0



# Getting 0 and 1

<sup>ONE</sup>  
~~ZERO~~(a):  
not a = NAND(a, a)  
return NAND(a, not a)

2

<sup>ZERO</sup>  
~~ONE~~(a):  
one = ONE(a)  
return NAND(one, one)

3

# Getting 0 and 1

```
def ONE(a):  
    not_a = NAND(a, a)  
    return NAND(a, not_a)
```

*3 gates*

```
def ZERO(a):  
    one = ONE(a)  
    return NAND(one, one)
```

# Computing any function

- Make a variable to represent each possible input
- Assign its value to match the correct output
- Use LOOKUP to select the proper output for the given input

# Straightline Code for F

```
def F(x0,x1,x2):
    F000=0 ZERO(x0)
    F001=0
    F010=1 ←
    F011=0
    F100=1
    F101=1
    F110=0
    F111=1 0
    return LOOKUP3(F000,F001,F010,F011,F100,F101,F110,F111,x0,x1,x2)
```

Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

$F()$ , AND()

101

# How many gates?

- How many gates does this construction take?

You can compute any finite function  $f: \{0,1\}^n \rightarrow \{0,1\}^m$  with a NAND Circuit using no more than  $c \cdot m \cdot 2^n$  gates

$$\frac{c \cdot m \cdot 2^n}{\nearrow}$$

Note: This can be improved to  $c \cdot m \cdot \frac{2^n}{n}$  (theorem 4.16 in TCS)

# Counting gates

1. Create variables for each input

$2^n$  variables

$$3 \cdot 2^n + 4 \cdot 2^n \cdot m$$

2. Assign 0,1 to each input

3 gates each,

$3 \cdot 2^n$  gates total

3. Do the LOOKUP

$m$  times

$m \cdot 3 \cdot 2^n$

$$4 \cdot 2^n \cdot m$$

$$3 \cdot 2^n + 4 \cdot 2^n \cdot m \leq \underbrace{C \cdot m \cdot 2^n}_{\neq}$$

$m \geq 1$

$$\cancel{2^n (3 + 4m)}$$

$$\begin{aligned} 3 \cdot 2^n + 4 \cdot 2^n \cdot m &\leq m \cdot 3 \cdot 2^n + 4 \cdot 2^n \cdot m \\ &\leq 2^n \cdot m (3 + 4) \end{aligned}$$

# What does this mean?

- Your laptop is a 64-bit machine. Given enough transistors, it can compute any function


$$f: \underline{\{0,1\}^{64}} \rightarrow \underline{\{0,1\}^{64}}$$

$$M \cdot 2^n$$
$$64 \cdot 2^{64}$$



# Any to Every



- Previous theorem:
    - We can compute ANY  $n$ -bit function using circuits/straightline programs
  - What we want:
    - A machine that can compute EVERY  $n$ -bit function
  - How do we do this?:
    - Define a function that "simulates" programs
    - Write a program that gives the same answer as a given program of  $n$  inputs,  $m$  outputs, and  $s$  lines
- 

# How are programs run?

- Have a table of variables
- Execute code in sequence
- Update values in table
- Return a value from the table

# Simulating XOR

```
def XOR(a,b):  
    3 u = NAND(a,b)  
    4 v = NAND(a, u)  
    5 w = NAND(b, u)  
    6 return NAND(v,w)
```

Variable	Value
a	0
b	1
u	1
v	1
w	0
return	1

# Simulating XOR

```
def XOR(a,b):  
    1 u = NAND(a,b) ←  
    2 v = NAND(a, u)  
    3 w = NAND(b, u)  
    4 return NAND(v,w)'
```

EVAL<sub>4,2,1</sub>

Variable	Value
a	0
b	1
u	1
v	1
w	0
return	1

# Defining EVAL

$$EVAL_{s,n,m}: \{0,1\}^{S(s)+n} \rightarrow \{0,1\}^m$$

program size  $\uparrow$   $\uparrow$   $\uparrow$   
inputs outputs

7, 2, 1  
11, 11, 11  
EVAL  
27, 8, 4

Input: bit string representing a program (first  $S(s)$  bits)  
plus input values (remaining  $n$  bits)

Output: the result of running the represented program  
on the provided input, or  $m$  0's if there's a "compile  
error"

# Programs as Bits

- To evaluate a program with another program, we need to convert the first program into bits
  1. Number each variable (first  $n$  go to input, last  $m$  to outputs)
  2. Represent each line as 3 numbers (outvar, in1, in2)
  3. Represent program as (n,m,[Lines])

```
def OR(a, b):  
    temp1 = NAND(a,a)      (2,0,0)  
    temp2 = NAND(b,b)      (3,1,1)  
    return NAND(temp1, temp2) (4,2,3)
```

(2,1,[(2,0,0),(3,1,1),(4,2,3)])

Variable	Number
a	0
b	1
temp1	2
temp2	3
return	4

# XOR to bits

```
def XOR(a,b):  
    u = NAND(a,b)  
    v = NAND(a, u)  
    w = NAND(b, u)  
    return NAND(v,w)
```

$n =$

$m =$

$s =$

Total bits =

Variable	Number

# XOR to bits

```
def XOR(a,b):  
    u = NAND(a,b)  
    v = NAND(a, u)  
    w = NAND(b, u)  
    return NAND(v,w)
```

$n = 2$

$m = 1$

$s = 4$

Variable	Number
a	0
b	1
u	2
v	3
w	4
return	5

Total bits = 3 [numbers per line] · 3 [bits per number] · 4[lines] + 6 [length of  $n + m$ ]



# How big is this?

1. Number each variable  $\lceil \log_2 3s \rceil$  bits each
2. Represent each line as 3 numbers (outvar, in1, in2)
3. Represent program as  $(\underbrace{n, m}_{2\lceil \log_2 s \rceil \text{ bits}}, [\text{Lines}])$   $3s \cdot \lceil \log_2 3s \rceil$  bits

$$S(s) \leq 4s \lceil \log_2 3s \rceil$$

$$\ell = \lceil \log_2 3s \rceil$$

# Defining EVAL

$$EVAL_{s,n,m}: \{0,1\}^{S(s)+n} \rightarrow \{0,1\}^m$$

Input: bit string representing a program (first  $S(s)$  bits)  
plus input values (remaining  $n$  bits)

Output: the result of running the represented program  
on the provided input, or  $m$  0's if there's a "compile  
error"

# Defining the EVAL function

$$n = 2$$
$$m = 1$$

Representation:

(2, 0, 1),  
(3, 0, 2),  
(4, 1, 2),  
(5, 3, 4)

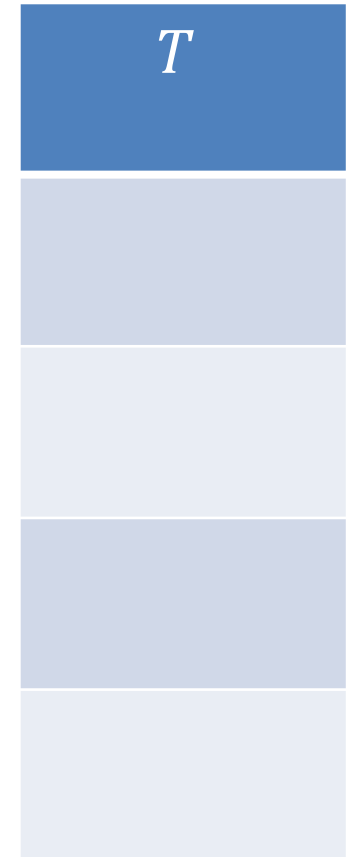
Input:

0, 1

Variable	Value
0	
1	
2	
3	
4	
5	

# Pseudocode for EVAL

- Table  $T$ :
  - holds variables and their values
- $GET(T, i)$ 
  - Returns the bit of  $T$  associated with variable  $i$
- $UPDATE(T, i, b)$ 
  - Returns a new table such that variable  $i$ 's value has been changed to  $b$



# Pseudocode for EVAL

- **Input:**

- Numbers  $n, m, s, t$   
representing the number of  
inputs, outputs, variables,  
and lines respectively
- $L$ , a list of triples  
representing the program
- A string  $x$  to be given as input  
to the program

- **Output:**

- Evaluation of the program  
represented by  $L$  when run  
on input  $x$

Let  $T$  be table of size  $t$

For  $i$  in range( $n$ ):

$T = \text{UPDATE}(T, i, x[i])$

For  $(i, j, k)$  in  $L$ :

$a = \text{GET}(T, j)$

$b = \text{GET}(T, k)$

$T = \text{UPDATE}(T, i, \text{NAND}(a, b))$

For  $i$  in range( $m$ ):

$Y[i] = \text{GET}(T, t - m + i)$

Return  $Y$

# EVAL in NAND

- Next we implement  $EVAL_{s,n,m}$  using NAND

# $GET(T, i)$

- Get the bit at “row”  $i$  of  $T$
- Look familiar?
- How many gates to implement?

# UPDATE

$$UPDATE_{\ell}: \{0,1\}^{s^{\ell}+\ell+1} \rightarrow \{0,1\}^{2^{\ell}}$$

- To change index  $i$  of table  $T$  to bit  $b$
- For every index except  $i$ , return the same value
- For index  $i$ , return  $b$  instead
- Define  $EQUAL_j: \{0,1\}^{\ell} \rightarrow \{0,1\}$  which returns 1 if the input binary number is equal to  $j$

Note:  $EQUAL_j$  can be done in  $c \cdot \ell$  gates



# UPDATE pseudocode

For  $j$  in range( $2^\ell$ ):

$a = EQUALS_j(i)$

$newT[j] = IF(a, b, T[j])$

Runs  $2^\ell$  times

Return  $newT$

# Conclusion