CS3102 Theory of Computation

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?

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 NAND-Straightline = NAND-Circ = AON-Circ = AON-straightline

Lookup

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

$$LOOKUP_k\colon \{0,1\}^{2^k+k} \to \{0,1\}$$

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

$$LOOKUP_k(x, i) = x_i$$

$LOOKUP_k$

$k = 3$

x:

0	1	0	0	1	1	1	0

First 2^k bits of input
Considered as a bitstring

i:

0	1	0

Last k bits of input
Considered as an index

Theorem

There is a NAND-Cricuit that computes $LOOKUP_k: \{0,1\}^{2^k+k} \rightarrow \{0,1\}$

Moreover, the number of gates required is at most $4 \cdot 2^k$

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 $k - 1$

x:

0	1	0	0	1	1	1	0

i:

0	1	0

x:

0	1	0	0

i:

1	0

Defining $LOOKUP_k$

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

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

Base Case

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

Next Step

LOOKUP2

```
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)

Counting Gates

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

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)

Idea

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

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

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

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

Getting 0 and 1

Getting 0 and 1

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

```python
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

How many gates?

- How many gates does this construction take?

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

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. Assign 0,1 to each input

3. Do the LOOKUP

What does this mean?

- Your laptop is a 64-bit machine. Given enough transistors, it can compute any function $f: \{0,1\}^{64} \to \{0,1\}^{64}$

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

Variable	Value

Simulating XOR

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

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

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)
    temp2 = NAND(b,b)
    return NAND(temp1, temp2)
```

(2,0,0)
(3,1,1)
(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] $\cdot$ 3 [bits per number] $\cdot$ 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) $3s \cdot \lceil \log_2 3s \rceil$ bits
3. Represent program as $(\underbrace{n,m,}_{2\lceil \log_2 s \rceil \text{ bits}}[\text{Lines}])$

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

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

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"

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

Defining the EVAL function

Representation:

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

Input:

0, 1

Variable	Value
0	
1	
2	
3	
4	
5	

Psuedocode 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

T

Psuedocode 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 = UPDATE(T, i, $x[i]$)
For (i,j,k) in L:
    a = GET(T, j)
    b = GET(T, k)
    T = UPDATE(T, i, NAND(a,b))
For i in range(m):
    $Y[i]$ = 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

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

- 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 $EQUAL_j: \{0,1\}^\ell \to \{0,1\}$ 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^{ℓ}):

$$\left.\begin{aligned} &a = EQUALS_j(i) \\ &newT[j] = IF(a, b, T[j]) \end{aligned}\right\} \text{Runs } 2^{\ell} \text{ times}$$

Return $newT$

Conclusion