Learning Objectives:

- Get comfort with syntactic sugar or automatic translation of higher level logic to NAND code.
- More techniques for translating informal or higher level language algorithms into NAND.
- Learn proof of major result: every finite function can be computed by some NAND program.
- Start thinking *quantitatively* about number of lines required for computation.

4

Syntactic sugar, and computing every function

> *"Syntactic sugar causes cancer of the semicolon."*, Alan Perlis, 1982.

The NAND programing language is pretty much as "bare bones" as programming languages come. After all, it only has a single operation. But, it turns out we can implement some "added features" on top of it. That is, we can show how we can implement those features using the underlying mechanisms of the language.

Let's start with a simple example. One of the most basic operations a programming language has is to assign the value of one variable into another. And yet in NAND, we cannot even do that, as we only allow assignments of the result of a NAND operation. Yet, it is possible to "pretend" that we have such an assignment operation, by transforming code such as

```
foo := bar
```

into the valid NAND code:

```
notbar := bar NAND bar
foo := notbar NAND notbar
```

the reason being that for every $a \in \{0,1\}$, $NAND(a,a) = NOT(aANDa) = NOT(a)$ and so in these two lines `notbar` is assigned the negation of `bar` and so `foo` is assigned the negation of the negation of `bar`, which is simply `bar`.

Thus in describing NAND programs we can (and will) allow ourselves to use the variable assignment operation, with the understanding that in actual programs we will replace every line of the first form with the two lines of the second form. In programming

language parlance this is known as "syntactic sugar", since we are not changing the definition of the language, but merely introducing some convenient notational shortcuts.[1] We will use several such "syntactic sugar" constructs to make our descriptions of NAND programs shorter and simpler. However, these descriptions are merely shorthand for the equivalent standard or "sugar free" NAND program that is obtained after removing the use of all these constructs. In particular, when we say that a function F has an s-line NAND program, we mean a standard NAND program, that does not use any syntactic sugar. The website http://www.nandpl.org contains an online "unsweetener" that can take a NAND program that uses these features and modifies it to an equivalent program that does not use them.

[1] This concept is also known as "macros" or "meta-programming" and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor. One modern example is the Babel JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a plug-in architecture, that allows users to add their own syntactic sugar to the language.

4.1 Some useful syntactic sugar

In this section, we will list some additional examples of "syntactic sugar" transformations. Going over all these examples can be somewhat tedious, but we do it for two reasons:

1. To convince you that despite its seeming simplicity and limitations, the NAND programming language is actually quite powerful and can capture many of the fancy programming constructs such as `if` statements and function definitions that exists in more fashionable languages.
2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course... `:)`

4.1.1 Constants

We can create variables `zero` and `one` that have the values 0 and 1 respectively by adding the lines

```
notx_0 := x_0 NAND x_0
one := x_0 NAND notx_0
zero := one NAND one
```

Note that since for every $x \in \{0,1\}$, $NAND(x,\overline{x}) = 1$, the variable `one` will get the value 1 regardless of the value of x_0, and the variable `zero` will get the value $NAND(1,1) = 0$.[2] Hence we can replace code such as `a := 0` with `a := one NAND one` and similarly `b := 1` will be replaced with `b:= zero NAND zero`.

[2] We could have saved a couple of lines using the convention that uninitialized variables default to 0, but it's always nice to be explicit.

4.1.2 *Conditional statements*

Another sorely missing feature in NAND is a conditional statement. We would have liked to be able to write something like

```
if (cond) {
   ...
  some code here
  ...
}
```

To ensure that there is code that will only be executed when the variable `cond` is equal to 1. We can do so by replacing every variable `foo` that is assigned a value in the code by a variable `tempfoo` and then simply execute the code (*regardless* of whether `cond` is false or true). After the code is executed, we want to ensure that if `cond` is true then the value of every such variable `foo` is replaced with `tempfoo`, and otherwise the value of `foo` should be unchanged.[3] We do so by assigning to every variable `foo` the value `MUX(foo,tempfoo,cond)` where $MUX : \{0,1\}^3 \to \{0,1\}$ is the *multiplexer* function that on input (a,b,c) outputs a if $c = 0$ and b if $c = 1$. This function has a 4-line NAND program:

[3] We assume here for simplicity of exposition that `cond` itself is not modified by the code inside the if statement. Otherwise, we will copy it to a temporary variable as well.

```
nx_2 := x_2 NAND x_2
u := x_0 NAND nx_2
v := x_1 NAND x_2
y_0 := u NAND v
```

We leave it as Exercise 4.2 to verify that this program does indeed compute the MUX function.

4.1.3 *Functions / Macros*

Another staple of almost any programming language is the ability to execute functions. However, we can achieve the same effect as (non recursive) functions using "copy pasting". That is, we can replace code such as

```
def a,b := Func(c,d) {
   function_code
}
...
e,f := Func(g,h)
```

with

```
...
function_code'
...
```

where `function_code'` is obtained by replacing all occurrences of `a` with `e`,`f` with `b`, `c` with `g`, `d` with `h`. When doing that we will need to ensure that all other variables appearing in `function_code'` don't interfere with other variables by replacing every instance of a variable `foo` with `upfoo` where `up` is some unique prefix.

4.1.4 *Bounded loops*

We can use "copy paste" to implement a bounded variant of *loops*, as long we only need to repeat the loop a fixed number of times. For example, we can use code such as:

```
for i in [7,9,12] do {
   y_i := foo_i NAND x_i
}
```

as shorthand for

```
y_7 := foo_7 NAND x_7
y_9 := foo_9 NAND x_9
y_12 := foo_12 NAND x_12
```

More generally, we will replace code of the form

```
for i in RANGE do {
   code
}
```

where `RANGE` specifies a finite set $I = \{i_0, \ldots, i_{k-1}\}$ of natural numbers, with $|R|$ copies of `code`, where for $j \in [k]$, we replace all occurences of `_i` in the j-th copy with `_`$\langle i_j \rangle$. We specify the set $I = \{i_0, \ldots, i_{k-1}\}$ by simply writing `[` $\langle i_0 \rangle$, $\langle i_1 \rangle$, ..., $\langle i_{k-1} \rangle$ `]`. We will also use the $\langle beg \rangle : \langle end \rangle$ notation so specify the interval $\{beg, beg + 1, \ldots, end - 1\}$. So for example

```
for i in [3:5,7:10] do {
   foo_i := bar_i NAND baz_i
}
```

will be a shorthand for

```
foo_3 := bar_3 NAND baz_3
foo_4 := bar_4 NAND baz_4
```

```
foo_7 := bar_7 NAND baz_7
foo_8 := bar_8 NAND baz_8
foo_9 := bar_9 NAND baz_9
```

One can also consider fancier versions, including inner loops and allowing arithmetic expressions such as `<5*i+3>` in indices. The crucial point is that (unlike most programming languages) we do not allow the number of times the loop is executed to depend on the input, and so it is always possible to "expand out" the loop by simply copying the code the requisite number of times.

4.1.5 *Example:*

Using these features, we can express the code of the ADD_2 function we saw last lecture as

```
def c := AND(a,b) {
  notc := a NAND b
  c := notc NAND notc
}
def c := XOR(a,b) {
   u := a NAND b
   v := a NAND u
   w := b NAND u
   c := v NAND w
}
y_0 := XOR(x_0,x_2) // add first digit
c_1 := AND(x_0,x_2)
z_1 := XOR(x_1,x_2) // add second digit
c_2 := AND(x_1,x_2)
y_1 := XOR(c_1,y_2) // add carry from before
c'_2 := AND(c_1,y_2)
y_2 := XOR(c'_2,x_2)
```

4.1.6 *More indices*

As stated, the NAND programming language only allows for "one dimensional arrays", in the sense that we can use variables such as `foo_7` or `foo_29` but not `foo_5,15`. However we can easily embed two dimensional arrays in one-dimensional ones using a one-to-one function $PAIR : \mathbb{N}^2 \to \mathbb{N}$. (For example, we can use $PAIR(x,y) = 2^x 3^y$, but there are also more efficient embeddings, see Exercise 4.1.)

Hence we can replace any variable of the form `foo_`$\langle i \rangle$`,`$\langle j \rangle$ with `foo_`$\langle PAIR(i,j) \rangle$, and similarly for three dimensional arrays.

4.1.7 Non-Boolean variables, lists and integers

While the basic variables in NAND++ are Boolean (only have 0 or 1), we can easily extend this to other objects using encodings. For example, we can encode the alphabet {`a`,`b`,`c`,`d`,`e`,`f` } using three bits as $000, 001, 010, 011, 100, 101$. Hence, given such an encoding, we could use the code

```
foo := "b"
```

would be a shorthand for the program

```
foo_0 := 0
foo_1 := 0
foo_2 := 1
```

Using our notion of multi-indexed arrays, we can also use code such as

```
foo := "be"
```

as a shorthand for

```
foo_{0,0} := 0
foo_{0,1} := 0
foo_{0,2} := 1
foo_{1,0} := 1
foo_{1,1} := 0
foo_{1,2} := 0
```

which can then in turn be mapped to standard NAND code using a one-to-one embedding $pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as above.

4.1.8 Storing integers

We can also handle non-finite alphabets, such as integers, by using some prefix-free encoding and encoding the integer in an array. For example, to store non-negative integers, we can use the convention that `01` stands for 0, `11` stands for 1, and `00` is the end marker. To store integers that could be potentially negative we can use the convention `10` in the first coordinate stands for the negative sign.[4] So, code such as

[4] This is just an arbitrary choice made for concreteness, and one can choose other representations. In particular, as discussed before, if the integers are known to have a fixed size, then there is no need for additional encoding to make them prefix-free.

```
foo := 5 // (1,0,1) in binary
```

will be shorthand for

```
foo_0 := 1
foo_1 := 1
foo_2 := 0
foo_3 := 1
foo_4 := 1
foo_5 := 1
foo_6 := 0
foo_7 := 0
```

while

```
foo := -5
```

will be the same as

```
foo_0 := 1
foo_1 := 0
foo_2 := 1
foo_3 := 1
foo_4 := 0
foo_5 := 1
foo_6 := 1
foo_7 := 1
foo_8 := 0
foo_9 := 0
```

Using multidimensional arrays, we can use arrays of integers and hence replace code such as

```
foo := [12,7,19,33]
```

with the equivalent NAND expressions.

For integer valued variables, we can use the standard algorithms of addition, multiplication, comparisons and so on.. to write code such as

```
j := k + l
if (m*n>k) {
   code...
}
```

which then gets translated into standard NAND++ program by copy pasting these algorithms.

4.2 *Adding and multiplying n bit numbers*

We have seen how to add one and two bit numbers. We can use the gradeschool algorithm to show that NAND programs can add n-bit numbers for every n:

Theorem 4.1 — Addition using NAND programs. For every n, let $ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{n+1}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the sum of the numbers that x and x' represent. Then there is a NAND program that computes the function ADD_n. Moreover, the number of lines in this program is smaller than $100n$.

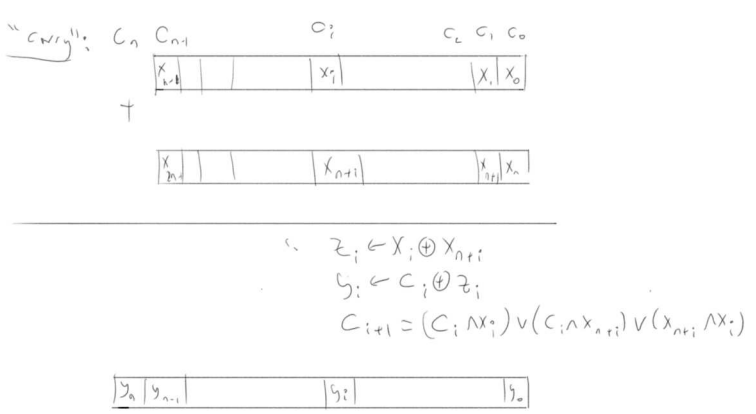

Figure 4.1: Translating the gradeschool addition algorithm into a NAND program. If at the i^{th} stage, the i^{th} digits of the two numbers are x_i and x_{n+i} and the carry is c_i, then the i^{th} digit of the sum will be $(x_i XOR x_{n+i}) XOR c_i$ and the new carry c_{i+1} will be equal to 1 if any two values among c_i, x_i, x_{n+i} are 1.

Proof. To prove this theorem we repeatedly appeal to the notion of composition, and to the "gradeschool" algorithm for addition. To add the numbers $(x_0, \ldots, x_{n-1})$ and $(x_n, \ldots, x_{2n-1})$, we set $c_0 = 0$ and do the following for $i = 0, \ldots, n-1$:

* Compute $z_i = XOR(x_i, x_{n+i})$ (add the two corresponding digits)

* Compute $y_i = XOR(z_i, c_i)$ (add in the carry to get the final digit)

* Compute $c_{i+1} = ATLEASTTWO(x_i, x_{n+i}, c_i)$ where $ATLEASTTWO : \{0,1\}^3 \rightarrow \{0,1\}$ is the function that maps (a, b, c) to 1 if $a + b + c \geq 2$. (The new carry is 1 if and only if at least two of the values x_i, x_{n+i}, y_i were equal to 1.) The most significant digit y_n of the output will of course be the last carry c_n.

To transform this algorithm to a NAND program we just need

to plug in the program for XOR, and use the observation (see Exercise 4.3) that

$$\begin{aligned} ATLEASTTWO(a,b,c) &= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\ &= NAND(NOT(NAND(NAND(a,b), NAND(a,c))), NAND(b,c)) \end{aligned} \tag{4.1}$$

We leave accounting for the number of lines, and verifying that it is smaller than $100n$, as an exercise to the reader. ■

See the website http://nandpl.org for an applet that produces, given n, a NAND program that computes ADD_n.[5]

[5] TODO: maybe add the example of the code of ADD_4? (using syntactic sugar)

4.2.1 *Multiplying numbers*

Once we have addition, we can use the gradeschool algorithm to obtain multiplication as well, thus obtaining the following theorem:

Theorem 4.2 — Multiplication NAND programs. For every n, let $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the product of the numbers that x and x' represent. Then there is a NAND program that computes the function $MULT_n$. Moreover, the number of lines in this program is smaller than $1000n^2$.

We omit the proof, though in Exercise 4.6 we ask you to supply a "constructive proof" in the form of a program (in your favorite programming language) that on input a number n, outputs the code of a NAND program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba's algorithm to show that there is a NAND program of $O(n^{\log_2 3})$ lines to compute $MULT_n$ (and one can even get further asymptotic improvements using the newer algorithms).

4.3 *Functions beyond arithmetic*

We have seen that NAND programs can add and multiply numbers. But can they compute other type of functions, that have nothing to do with arithmetic? Here is one example:

Definition 4.3 — Lookup function. For every k, the *lookup* function $LOOKUP_k : \{0,1\}^{2^k+k} \to \{0,1\}$ is defined as follows: For every

$x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x,i) = x_i \tag{4.2}$$

where x_i denotes the i^{th} entry of x, using the binary representation to identify i with a number in $\{0,\ldots,2^k-1\}$.

The function $LOOKUP_1 : \{0,1\}^3 \to \{0,1\}$ maps $(x_0,x_1,i) \in \{0,1\}^3$ to x_i. It is actually the same as the MUX function we have seen above, that has a 4 line NAND program. However, can we compute higher levels of $LOOKUP$? This turns out to be the case:

Theorem 4.4 — Lookup function. For every k, there is a NAND program that computes the function $LOOKUP_k : \{0,1\}^{2^k+k} \to \{0,1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

4.3.1 *Constructing a NAND program for LOOKUP*

We now prove Theorem 4.4. We will do so by induction. That is, we show how to use a NAND program for computing $LOOKUP_k$ to compute $LOOKUP_{k+1}$. Let us first see how we do this for $LOOKUP_2$. Given input $x = (x_0,x_1,x_2,x_3)$ and an index $i = (i_0,i_1)$, if the most significant bit i_1 of the index is 0 then $LOOKUP_2(x,i)$ will equal x_0 if $i_0 = 0$ and equal x_1 if $i_0 = 1$. Similarly, if the most significant bit i_1 is 1 then $LOOKUP_2(x,i)$ will equal x_2 if $i_0 = 0$ and will equal x_3 if $i_0 = 1$. Another way to say this is that

$$LOOKUP_2(x_0,x_1,x_2,x_3,i_0,i_1) = LOOKUP_1(LOOKUP_1(x_0,x_1,i_0),LOOKUP_1(x_2,x_3,i_0),i_1) \tag{4.3}$$

That is, we can compute $LOOKUP_2$ using three invocations of $LOOKUP_1$. The "pseudocode" for this program will be

```
z_0 := LOOKUP_1(x_0,x_1,x_4)
z_1 := LOOKUP_1(x_2,x_3,x_4)
y_0 := LOOKUP_1(z_0,z_1,x_5)
```

(Note that since we call this function with $(x_0,x_1,x_2,x_3,i_0,i_1)$, the inputs `x_4` and `x_5` correspond to i_0 and i_1.) We can obtain an actual "sugar free" NAND program of at most 12 lines by replacing the calls to `LOOKUP_1` by an appropriate copy of the program above.

We can generalize this to compute $LOOKUP_3$ using two invocations of $LOOKUP_2$ and one invocation of $LOOKUP_1$. That is, given input $x = (x_0,\ldots,x_7)$ and $i = (i_0,i_1,i_2)$ for $LOOKUP_3$, if the most significant bit of the index i_2 is 0, then the output of $LOOKUP_3$ will

equal $LOOKUP_2(x_0, x_1, x_2, x_3, i_0, i_1)$, while if this index i_2 is 1 then the output will be $LOOKUP_2(x_4, x_5, x_6, x_7, i_0, i_1)$, meaning that the following pseudocode can compute $LOOKUP_3$,

```
z_0 := LOOKUP_2(x_0,x_1,x_2,x_3,x_8,x_9)
z_1 := LOOKUP_2(x_4,x_5,x_6,x_7,x_8,x_9)
y_0 := LOOKUP_1(z_0,z_1,x_10)
```

where again we can replace the calls to `LOOKUP_2` and `LOOKUP_1` by invocations of the process above.

Formally, we can prove the following lemma:

Lemma 4.5 — Lookup recursion. For every $k \geq 2$, $LOOKUP_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$ is equal to

$$LOOKUP_1(LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-2}), LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-2}), i_{k-1}) \tag{4.4}$$

Proof. If the most significant bit i_{k-1} of i is zero, then the index i is in $\{0, \ldots, 2^{k-1} - 1\}$ and hence we can perform the lookup on the "first half" of x and the result of $LOOKUP_k(x, i)$ will be the same as $a = LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-1})$. On the other hand, if this most significant bit i_{k-1} is equal to 1, then the index is in $\{2^{k-1}, \ldots, 2^k - 1\}$, in which case the result of $LOOKUP_k(x, i)$ is the same as $b = LOOKUP_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$. Thus we can compute $LOOKUP_k(x, i)$ by first computing a and b and then outputting $LOOKUP_1(a, b, i_{k-1})$. ■

Lemma 4.5 directly implies Theorem 4.4. We prove by induction on k that there is a NAND program of at most $4 \cdot 2^k$ lines for $LOOKUP_k$. For $k = 1$ this follows by the four line program for $LOOKUP_1$ we've seen before. For $k > 1$, we use the following pseudocode

```
a = LOOKUP_(k-1)(x_0,...,x_(2^(k-1)-1),i_0,...,i_(k-2))
b = LOOKUP_(k-1)(x_(2^(k-1)),...,x_(2^(k-1),i_0,...,i_(k-2))
y_0 = LOOKUP_1(a,b,i_{k-1})
```

If we let $L(k)$ be the number of lines required for $LOOKUP_k$, then the above shows that

$$L(k) \leq 2L(k-1) + 4 \,. \tag{4.5}$$

We will prove by induction that $L(k) \leq 4(2^k - 1)$. This is true for $k = 1$ by our construction. For $k > 1$, using the inductive hypothesis and Eq. (4.5), we get that

$$L(k) \leq 2 \cdot 4 \cdot (2^{k-1} - 1) + 4 = 4 \cdot 2^k - 8 + 4 = 4(2^k - 1) \tag{4.6}$$

completing the proof of Theorem 4.4.

4.4 *Computing* every *function*

At this point we know the following facts about NAND programs:

1. They can compute at least some non trivial functions.
2. Coming up with NAND programs for various functions is a very tedious task.

Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND programs can compute *every* finite function:

Theorem 4.6 — Universality of NAND. For every n, m and function $F : \{0,1\}^n \to \{0,1\}^m$, there is a NAND program that computes the function F. Moreover, there is such a program with at most $O(m2^n)$ lines.

The implicit constant in the $O(\cdot)$ notation can be shown to be at most 10. We also note that the bound of Theorem 4.6 can be improved to $O(m2^n/n)$, see Remark 4.4.1.

4.4.1 *Proof of NAND's Universality*

To prove Theorem 4.6, we need to give a NAND program for *every* possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). In Exercise 4.8 you will show how to extend the proof for all values of m. A function $F : \{0,1\}^n \to \{0,1\}$ can be specified by a table of its values for each one of the 2^n inputs. For example, the table below describes one particular function $G : \{0,1\}^4 \to \{0,1\}$:[6]

[6] In case you are curious, this is the function that computes the digits of π in the binary basis.

We can see that for every $x \in \{0,1\}^4$, $G(x) = LOOKUP_4(1100100100001111, x)$. Therefore the following is NAND "pseudocode" to compute G:

```
G0000 := 1
G0001 := 1
G0010 := 0
G0011 := 0
G0100 := 1
G0101 := 0
```

Input (x)	Output ($G(x)$)
0000	1
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

```
G0110 := 0
G0111 := 1
G1000 := 0
G1001 := 0
G1010 := 0
G1011 := 0
G1100 := 1
G1101 := 1
G1110 := 1
G1111 := 1
y_0 := LOOKUP(G0000,G0001,G0010,G0011,G0100,
          G0101,G0110,G0111,G1000,G1001,
          G1010,G1011,G1100,G1101,G1111,
          x_0,x_1,x_2,x_3)
```

Recall that we can translate this pseudocode into an actual NAND program by adding three lines to define variables `zero` and `one` that are initialized to 0 and 1 repsectively, and then replacing a statement such as `Gxxx := 0` with `Gxxx := one NAND one` and a statement such as `Gxxx := 1` with `Gxxx := zero NAND zero`. The call to `LOOKUP` will be replaced by the NAND program that computes $LOOKUP_4$, but we will replace the variables `i_0`,...,`i_3` in this program with `x_0`,...,`x_3` and the variables `x_0`,...,`x_15` with `G000`, ..., `G1111`.

There was nothing about the above reasoning that was particular

to this program. Given every function $F : \{0,1\}^n \to \{0,1\}$, we can write a NAND program that does the following:

1. Initialize 2^n variables of the form `F00...0` till `F11...1` so that for every $z \in \{0,1\}^n$, the variable corresponding to z is assigned the value $F(z)$.

2. Compute $LOOKUP_n$ on the 2^n variables initialized in the previous step, with the index variable being the input variables `x_`$\langle 0 \rangle$,...,`x_`$\langle 2^n - 1 \rangle$. That is, just like in the pseudocode for `G` above, we use `y_0 := LOOKUP(F00..00,F00...01,...,F11..1,x_0,..,x_`$\langle n - 1 \rangle$`)`

The total number of lines in the program will be 2^n plus the $4 \cdot 2^n$ lines that we pay for computing $LOOKUP_n$. This completes the proof of Theorem 4.6.

The NAND programming language website allows you to construct a NAND program for an arbitrary function.

R **Advanced note: improving by a factor of n** By being a little more careful, we can improve the bound of Theorem 4.6 and show that every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n/n)$ lines. As before, it is enough to prove the case that $m = 1$.

The idea is to use the technique known as *memoization*. Let $k = \log(n - 2\log n)$ (the reasoning behind this choice will become clear later on). For every $a \in \{0,1\}^{n-k}$ we define $F_a : \{0,1\}^k \to \{0,1\}$ to be the function that maps $w_0,\ldots,w_{k-1}$ to $F(a_0,\ldots,a_{n-k-1},w_0,\ldots,w_{k-1})$. On input $x = x_0,\ldots,x_{n-1}$, we can compute $F(x)$ as follows: First we compute a 2^{n-k} long string P whose a^{th} entry (identifying $\{0,1\}^{n-k}$ with $[2^{n-k}]$) equals $F_a(x_{n-k},\ldots,x_{n-1})$. One can verify that $F(x) = LOOKUP_{n-k}(P,x_0,\ldots,x_{n-k-1})$. Since we can compute $LOOKUP_{n-k}$ using $O(2^{n-k})$ lines, if we can compute the string P (i.e., compute variables `P_`$\langle 0 \rangle,\ldots,$ `P_`$\langle 2^{n-k} - 1 \rangle$) using T lines, then we can compute F in $O(2^{n-k}) + T$ lines. The trivial way to compute the string P would be to use $O(2^k)$ lines to compute for every a the map $x_0,\ldots,x_{k-1} \mapsto F_a(x_0,\ldots,x_{k-1})$ as in the proof of Theorem 4.6. Since there are 2^{n-k} a's, that would be a total cost of $O(2^{n-k} \cdot 2^k) = O(2^n)$ which would not improve at all on the bound of Theorem 4.6. However, a more careful observation shows that we are making some *redundant* computations. After all, there are only 2^{2^k} distinct functions mapping k bits to one bit. If a and a' satisfy that $F_a = F_{a'}$

then we don't need to spend 2^k lines computing both $F_a(x)$ and $F_{a'}(x)$ but rather can only compute the variable $\texttt{P\_}\langle a \rangle$ and then copy $\texttt{P\_}\langle a \rangle$ to $\texttt{P\_}\langle a' \rangle$ using $O(1)$ lines. Since we have 2^{2^k} unique functions, we can bound the total cost to compute P by $O(2^{2^k}2^k) + O(2^{n-k})$. Now it just becomes a matter of calculation. By our choice of k, $2^k = n - 2\log n$ and hence $2^{2^k} = \frac{2^n}{n^2}$. Since $n/2 \leq 2^k \leq n$, we can bound the total cost of computing $F(x)$ (including also the additional $O(2^{n-k})$ cost of computing $LOOKUP_{n-k}$) by $O(\frac{2^n}{n^2} \cdot n) + O(2^n/n)$, which is what we wanted to prove.

Discussion: In retrospect, it is perhaps not surprising that every finite function can be computed with a NAND program. A finite function $F : \{0,1\}^n \to \{0,1\}^m$ can be represented by simply the list of its outputs for each one of the 2^n input values. So it makes sense that we could write a NAND program of similar size to compute it. What is more interesting is that *some* functions, such as addition and multiplication, have a much more efficient representation: one that only requires $O(n^2)$ or even smaller number of lines.

4.5 The class $SIZE_{n,m}(T)$

For every $n, m, T \in \mathbb{N}$, we denote by $SIZE_{n,m}(T)$, the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$ that can be computed by NAND programs of at most T lines. Theorem 4.6 shows that $SIZE_{n,m}(4m2^n)$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. The results we've seen before can be phrased as showing that $ADD_n \in SIZE_{2n,n+1}(100n)$ and $MULT_n \in SIZE_{2n,2n}(10000n^{\log_2 3})$.[7]

[7] TODO: check constants

4.6 Lecture summary

- We can define the notion of computing a function via a simplified "programming language", where computing a function F in T steps would correspond to having a T-line NAND program that computes F.
- While the NAND programming only has one operation, other operations such as functions and conditional execution can be implemented using it.

- Every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n)$ lines (and in fact at most $O(m2^n/n)$ lines).
- Sometimes (or maybe always?) we can translate an *efficient* algorithm to compute F into a NAND program that computes F with a number of lines comparable to the number of steps in this algorithm.

4.7 Exercises

Exercise 4.1 — Pairing. 1. Prove that the map $F(x,y) = 2^x 3^y$ is a one-to-one map from $\mathbb{N}^2$ to $\mathbb{N}$.

2. Show that there is a one-to-one map $F : \mathbb{N}^2 \to \mathbb{N}$ such that for every x, y, $F(x,y) \leq 100 \cdot \max\{x,y\}^2 + 100$.

3. For every k, show that there is a one-to-one map $F : \mathbb{N}^k \to \mathbb{N}$ such that for every $x_0, \ldots, x_{k-1} \in \mathbb{N}$, $F(x_0, \ldots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \ldots + x_{k-1} + 100k)^k$.

■

Exercise 4.2 — Computing MUX. Prove that the NAND program below computes the function MUX (or $LOOKUP_1$) where $MUX(a,b,c)$ equals a if $c = 0$ and equals b if $c = 1$: ■

```
nx_2 := x_2 NAND x_2
u := x_0 NAND nx_2
v := x_1 NAND x_2
y_0 := u NAND v
```

Exercise 4.3 — At least two. Give a NAND program of at most 6 lines to compute $ATLEASTTWO : \{0,1\}^3 \to \{0,1\}$ where $ATLEASTTWO(a,b,c) = 1$ iff $a + b + c \geq 2$. ■

Exercise 4.4 — Conditional statements. In this exercise we will show that even though the NAND programming language does not have an `if .. then .. else ..` statement, we can still implement it. Suppose that there is an s-line NAND program to compute $F : \{0,1\}^n \to \{0,1\}$ and an s'-line NAND program to compute $F' : \{0,1\}^n \to \{0,1\}$. Prove that there is a program of at most $s + s' + 10$ lines to compute the function $G : \{0,1\}^{n+1} \to \{0,1\}$ where $G(x_0, \ldots, x_{n-1}, x_n)$ equals $F(x_0, \ldots, x_{n-1})$ if $x_n = 0$ and equals $F'(x_0, \ldots, x_{n-1})$ otherwise. ■

Exercise 4.5 — Addition. Write a program using your favorite programming language that on input an integer n, outputs a NAND program

that computes ADD_n. Can you ensure that the program it outputs for ADD_n has fewer than $10n$ lines? ■

Exercise 4.6 — Multiplication. Write a program using your favorite programming language that on input an integer n, outputs a NAND program that computes $MULT_n$. Can you ensure that the program it outputs for $MULT_n$ has fewer than $1000 \cdot n^2$ lines? ■

Exercise 4.7 — Efficient multiplication (challenge). Write a program using your favorite programming language that on input an integer n, outputs a NAND program that computes $MULT_n$ and has at most $10000n^{1.9}$ lines.[8] What is the smallest number of lines you can use to multiply two 2048 bit numbers? ■

[8] **Hint:** Use Karatsuba's algorithm

Exercise 4.8 — Multibit function. Prove that

a. If there is an s-line NAND program to compute $F : \{0,1\}^n \to \{0,1\}$ and an s'-line NAND program to compute $F' : \{0,1\}^n \to \{0,1\}$ then there is an $s + s'$-line program to compute the function $G : \{0,1\}^n \to \{0,1\}^2$ such that $G(x) = (F(x), F'(x))$.

b. For every function $F : \{0,1\}^n \to \{0,1\}^m$, there is a NAND program of at most $10m \cdot 2^n$ lines that computes F. ■

4.8 Bibliographical notes

4.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

(to be completed)

4.10 Acknowledgements