Learning Objectives:

- See that computation can be precisely modeled.
- Learn the NAND computational model.
- Comfort switching between description of NAND programs as *code* and as *tuples*.
- Begin acquiring skill of translating informal algorithms into NAND code.

3

Defining computation

> *"there is no reason why mental as well as bodily labor should not be economized by the aid of machinery"*, Charles Babbage, 1852

> *"If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results."*, Charles Babbage, 1864

> *"To understand a program you must become both the machine and the program."*, Alan Perlis, 1982

People have been computing for thousands of years, with aids that include not just pen and paper, but also abacus, slide rulers, various mechanical devices, and modern electronic computers. A priori, the notion of computation seems to be tied to the particular mechanism that you use. You might think that the "best" algorithm for multiplying numbers will differ if you implement it in *Python* on a modern laptop than if you use pen and paper. However, as we saw in the introduction, an algorithm that is asymptotically better would eventually beat a worse one regardless of the underlying technology. This gives us hope for a *technology independent* way of defining computation, which is what we will do in this lecture.

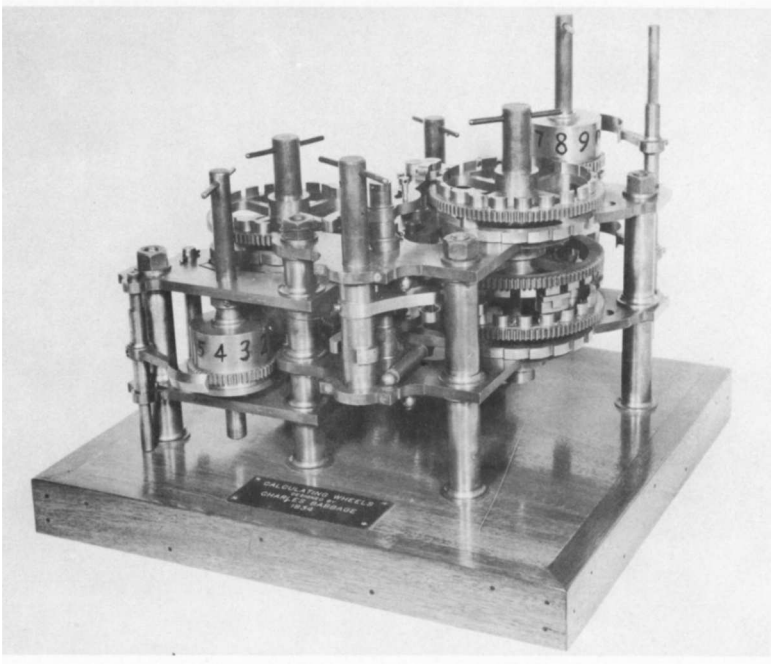

Figure 3.1: Calculating wheels by Charles Babbage. Image taken from the Mark I 'operating manual'

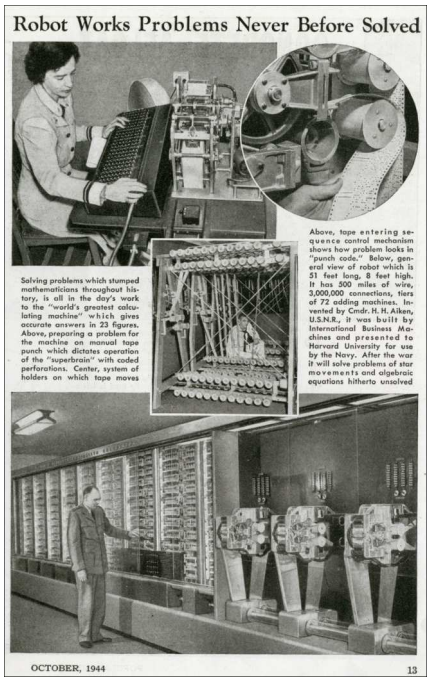

Figure 3.2: A 1944 *Popular Mechanics* article on the Harvard Mark I computer.

3.1 *Defining computation*

The name "algorithm" is derived from the Latin transliteration of Muhammad ibn Musa al-Khwarizmi, who was a Persian scholar during the 9th century whose books introduced the western world to the decimal positional numeral system, as well as the solutions of linear and quadratic equations (see Fig. 3.4). Still his description of the algorithms were rather informal by today's standards. Rather than use "variables" such as x, y, he used concrete numbers such as 10 and 39, and trusted the reader to be able to extrapolate from these examples.[1]

[1] Indeed, extrapolation from examples is still the way most of us first learn algorithms such as addition and multiplication, see Fig. 3.3)

Here is how al-Khwarizmi described how to solve an equation of the form $x^2 + bx = c$:[2]

> *[How to solve an equation of the form] "roots and squares are equal to numbers": For instance "one square , and ten roots of the same, amount to thirty-nine dirhems" that is to say, what must be the square which, when increased by ten of its own root, amounts to thirty-nine? The solution is this: you halve the number of the roots, which in the present instance yields five. This you multiply by itself; the product is twenty-five. Add this to thirty-nine' the sum is sixty-four. Now take the root of this, which is eight, and subtract from it half the number of roots, which is five; the remainder is three. This is the root of the square which you sought for; the square itself is nine.*

[2] Translation from "The Algebra of Ben-Musa", Fredric Rosen, 1831.

For the purposes of this course, we will need a much more precise way to define algorithms. Fortunately (or is it unfortunately?), at least at the moment, computers lag far behind school-age children in learning from examples. Hence in the 20th century people have come up with exact formalisms for describing algorithms, namely *programming languages*. Here is al-Khwarizmi's quadratic equation solving algorithm described in the Python programming language:[3]

[3] For concreteness we will sometimes include code of actual programming languages in these notes. However, these will be simple enough to be understandable even by people that are not familiar with these languages.

```
def solve_eq(b,c):
    # return solution of x^2 + bx = c using Al Khwarizmi's
        instructions
    val1 = b/2.0 # halve the number of the roots
    val2 = val1*val1 # this you multiply by itself
    val3 = val2 + c # Add this to thirty-nine (c)
    val4 = math.sqrt(val3) # take the root of this
    val5 = val4 - val1 # subtract from it half the number of
        roots
```

Figure 3.3: An explanation for children of the two digit addition algorithm

Figure 3.4: Text pages from Algebra manuscript with geometrical solutions to two quadratic equations. Shelfmark: MS. Huntingdon 214 fol. 004v-005r

```
    return val5 # This is the root of the square which you
        sought for
```

3.2 *The NAND Programming language*

We can try to use a modern programming language such as Python or C for our formal model of computation, but it would be quite hard to reason about, given that the Python language reference has more than 100 pages. Thus we will define computation using an extremely simple "programming language": one that has only a single operation. This raises the question of whether this language is rich enough to capture the power of modern computing systems. We will see that (to a first approximation), the answer to this question is **Yes**.

We start by defining a programming language that can only compute *finite* functions. That is, functions F that map $\{0,1\}^n$ to $\{0,1\}^m$ for some natural numbers m, n. Later we will discuss how to extend the language to allow for a single program that can compute a function of every length, but the finite case is already quite interesting and will give us a simple setting for exploring some of the salient features of computing.

The *NAND programming language* has no loops, functions, or if statements. It has only a single operation: `NAND`. That is, every line in a NAND program has the form:

```
foo := bar NAND baz
```

where `foo`, `bar`, `baz` are variable names.[4] When this line is executed, the variable `foo` is assigned the *negation of the logical AND* of (i.e., the NAND operation applied to) the values of the two variables `bar` and `baz`.[5]

All variables in the NAND programming language are *Boolean*: can take values that are either zero or one. Variables such as `x_22` or `y_18` (that is, of the form `x_`$\langle i \rangle$ or `y_`$\langle i \rangle$ where i is a natural number) have a special meaning.[6] The variables beginning with `x_` are *input* variables and those beginning with `y_` are *output* variables. Thus for example the following four line NAND program takes an input of two bits and outputs a single bit:

```
u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
y_0 := v NAND w
```

[4] The terms `foo` and `bar` are often used to describe generic variable names in the context of programming, and we will follow this convention throughout the course. See the appendix and the website http://nandpl.org for a full specification of the NAND programming language.

[5] The *logical AND* of two bits $x, x' \in \{0,1\}$ is equal to 1 if $x = x' = 1$ and is equal to 0 otherwise. Thus its negation satisfies $NAND(0,0) = NAND(0,1) = NAND(1,0) = 1$, while $NAND(1,1) = 0$. If a variable hasn't been assigned a value, then its default value is zero.

[6] In these lecture notes, we use the convention that when we write $\langle e \rangle$ then we mean the numerical value of this expression. So for example if $k = 10$ then we can write `x_`$\langle k+7 \rangle$ to mean `x_17`. This is just for the notes: in the NAND programming language itself the indices have to be absolute numerical constants.

P Can you guess what function from $\{0,1\}^2$ to $\{0,1\}$ this program computes? It might be a good idea for you to pause here and try to figure this out.

To find the function that this program computes, we can run it on all the four possible two bit inputs: 00,01,10, and 11.

For example, let us consider the execution of this program on the input 00, keeping track of the values of the variables as the program runs line by line. On the website http://nandpl.org we can run NAND programs in a "debug" mode, which will produce an *execution trace* of the program.[7] When we run the program above on the input 01, we get the following trace:

[7] At present the web interface is not yet implemented, and you can run NAND program using an OCaml interpreter that you can download from that website. The implementation is in a fluid state and so the text below might not exactly match the output of the interpreter.

```
Executing step 1: "u␣␣␣:=␣x_0␣NAND␣x_1" x_0 = 0, x_1 = 1, u
    is assigned 1,
Executing step 2: "v␣␣␣:=␣x_0␣NAND␣u" x_0 = 0, u = 1, v is
    assigned 1,
Executing step 3: "w␣␣␣:=␣x_1␣NAND␣u" x_1 = 1, u = 1, w is
    assigned 0,
Executing step 4: "y_0␣:=␣v␣␣␣NAND␣w" v = 1, w = 0, y_0 is
    assigned 1,
Output is y_0=1
```

On the other hand if we execute this program on the input 11, then we get the following execution trace:

```
Executing step 1: "u␣␣␣:=␣x_0␣NAND␣x_1" x_0 = 1, x_1 = 1, u
    is assigned 0,
Executing step 2: "v␣␣␣:=␣x_0␣NAND␣u" x_0 = 1, u = 0, v is
    assigned 1,
Executing step 3: "w␣␣␣:=␣x_1␣NAND␣u" x_1 = 1, u = 0, w is
    assigned 1,
Executing step 4: "y_0␣:=␣v␣␣␣NAND␣w" v = 1, w = 1, y_0 is
    assigned 0,
Output is y_0=0
```

You can verify that on input 10 the program will also output 1, while on input 00 it will output zero. Hence the output of this program on every input is summarized in the following table:

In other words, this program computes the *exclusive or* (also known as XOR) function.

Input	Output
00	0
01	1
10	1
11	0

3.2.1 *Adding one-bit numbers*

Now that we can compute XOR, let us try something just a little more ambitious: adding a pair of one-bit numbers. That is, we want to compute the function $ADD_1 : \{0,1\}^2 \to \{0,1\}^2$ such that $ADD(x_0, x_1)$ is the binary representation of the addition of the two numbers x_0 and x_1. Since the sum of two $0/1$ values is a number in $\{0,1,2\}$, the output of the function ADD_1 is of length two bits.

If we write the sum $x_0 + x_1$ as $y_0 2^0 + y_1 2^1$ then the table of values for ADD_1 is the following:

Input		Output	
x_0	x_1	y_0	y_1
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

One can see that `y_0` will be the XOR of `x_0` and `x_1` and `y_1` will be the AND of `x_0` and `x_1`.[8] Thus we can compute one bit variable addition using the following program:

```
// Add two single-bit numbers
u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
y_0 := v NAND w
y_1 := u NAND u
```

If we run this program on the input $(1,1)$ we get the execution trace

```
Executing step 1: "u␣␣␣:=␣x_0␣NAND␣x_1" x_0 = 1, x_1 = 1, u
    is assigned 0,
Executing step 2: "v␣␣␣:=␣x_0␣NAND␣u" x_0 = 1, u = 0, v is
    assigned 1,
Executing step 3: "w␣␣␣:=␣x_1␣NAND␣u" x_1 = 1, u = 0, w is
    assigned 1,
```

[8] This is a special case of the general rule that when you add two digits $x, x' \in \{0,1,\ldots,b-1\}$ over the b-ary basis (in our case $b = 2$), then the output digit is $x + x' \pmod b$ and the carry digit is $\lfloor (x+x')/b \rfloor$.

```
Executing step 4: "y_0 := v NAND w" v = 1, w = 1, y_0 is
    assigned 0,
Executing step 5: "y_1 := u NAND u" u = 0, u = 0, y_1 is
    assigned 1,
Output is y_0=0, y_1=1
```

and so you can see that the output $(0,1)$ is indeed the binary encoding of $1+1=2$.

3.2.2 *Formal definitions*

For a NAND program P, its *input length* is the largest number n such that P contains a variable of the form $\texttt{x\_}\langle n-1\rangle$. P's *output length* is the largest number m such that P contains a variable of the form $\texttt{y\_}\langle m-1\rangle$.[9] Intuitively, if P is a NAND program with input length n and output length m, and $F : \{0,1\}^n \to \{0,1\}^m$ is some function, then P computes F if for every $x \in \{0,1\}^n$ and $y = F(x)$, whenever P is executed with the $\texttt{x\_}\langle i\rangle$ variable initialized to x_i for all $i \in [n]$, at the end of the execution the variable $\texttt{y\_}\langle j\rangle$ will equal y_j for all $j \in [m]$.

[9] As mentioned in the appendix, we require that all output variables are assigned a value, and that the largest index used in an s line NAND program is smaller than s. In particular this means that an s line program can have at most s inputs and outputs.

To make sure we have a precise and unambiguous definition of computation, we will now model NAND programs using sets and tuples, and recast the notion of computing a function in these terms.

Definition 3.1 — NAND program. A *NAND program* is a 4-tuple $P = (V, X, Y, L)$ of the following form:

- V (called the *variables*) is some finite set.
- X (called the *input variables*) is a tuple of elements in V, i.e. $X = (X_0, X_1, \ldots, X_{n-1})$ for some $n \in N$ where $X_i \in V$ for all $i \in [n]$. We require that the elements of X are distinct: $X_i \neq X_j$ for all $i \neq j$ in $[n]$.
- Y (called the *output variables*) is a tuple of elements in V, i.e., $Y = (Y_0, \ldots, Y_{m-1})$ for some $m \in \mathbb{N}$ where $Y_j \in V$ for all $j \in [m]$. We require that the elements of Y are distinct (i.e., $Y_i \neq Y_j$ for all $i \neq j$ in $[m]$) and that they are disjoint from X (i.e., $Y_i \neq X_j$ for every $i \in [n]$ and $j \in [m]$).
- L (called the *lines*) is a tuple of *triples* of V, i.e., $L \in (V \times V \times V)^*$. Intuitively, if the ℓ-th element of L is a triple (u, v, w) then this corresponds to the ℓ-th line of the program being u `:=` v `NAND` w. We require that for every triple (u, v, w), u does not appear in X and v, w do not appear in Y. Moreover, we require that

for every $v \in V$ (i.e., a member of V that is not equal to X_i for some i), v is contained in some triple in L.

The *number of inputs* of $P = (V, X, Y, Z)$ is equal to $|X|$ and the *number of outputs* is equal to $|Y|$.

This definition is somewhat long and cumbersome, but really corresponds to a straightforward modelling of NAND programs, under the map that V is the set of all variables appearing in the program, X corresponds to the tuple (`x_`$\langle 0 \rangle$, `x_`$\langle 1 \rangle$, ..., `x_`$\langle n-1 \rangle$), Y corresponds to the tuple (`y_`$\langle 0 \rangle$, `y_`$\langle 1 \rangle$, ..., `y_`$\langle m-1 \rangle$,) and L corresponds to the list of triples of the form (`foo , bar, , baz`) for every line `foo := bar NAND baz` in the program. Please pause here and verify that you understand this correspondence.

For example, one representation of the XOR program we described above is $P = (V, X, Y, L)$ where

- $V =$ { `x_0, x_1, v, u, w, y_0` }
- $X =$ (`x_0, x_1`)
- $Y =$ (`y_0`)
- $L =$ ((`u,x_0,x_1`), (`v,x_0,u`), (`w,x_1,u`), (`y_0,v,w`))

But since we have the freedom of choosing arbitrary sets for our variables, we can also represent the same program as (for example) $P' = (V', X', Y', L')$ where

- $V' = \{0, 1, 2, 3, 4, 5\}$
- $X' = (0, 1)$
- $Y' = (5)$
- $L' = ((3, 0, 1), (2, 0, 3), (4, 1, 3), (5, 2, 4))$

3.2.3 *Computing a function: formal definition*

Now that we defined NAND programs formally, we turn to formally defining the notion of computing a function. Before we do that, we will need to talk about the notion of the *configuration* of a NAND program. Such a configuration simply corresponds to the current line that is executed and the current values of all variables at a certain point in the execution. Thus we will model it as a pair (ℓ, σ) where ℓ

is a number between 0 and the total number of lines in the program, and σ maps every variable to its current value.[10] The initial configuration has the form $(0, \sigma_0)$ where 0 corresponds to the first line, and σ_0 is the assignment of zeroes to all variables and x_i's to the input variables. The final configuration will have the form (s, σ_s) where s is the number of lines (i.e., corresponding to "going past" the final line) and σ_s is the final values assigned to all variables, which in particular encodes also the values of the output variables.

[10] The number ℓ can be thought of as the "program counter" and refers to the line that is just about to be executed, when we number the lines from 0 till $s-1$ for some $s \in \mathbb{N}$. The program counter starts at 0, and after executing the last line (i.e., line number $s-1$), it equals s.

For example, if we run the XOR program about on the input `11` then the configuration of the program evolves as follows:

```
                  x_0 x_1 v u w y_0
0. u := x_0 NAND x_1 : 0 1 0 0 0 0
1. v := x_0 NAND u : 0 1 0 1 0 0
2. w := x_1 NAND u : 0 1 1 1 0 0
3. y_0 := v NAND w : 0 1 1 1 0 0
4. (after halting) : 0 1 1 1 0 1
```

We now write the formal definition. As always, it is a good practice to verify that this formal definition matches the intuitive description above:

Definition 3.2 — Configuration of a NAND program. Let $P = (V, X, Y, L)$ be a NAND program, and let $n = |X|$, $m = |Y|$ and $s = |L|$. A *configuration* of P is a pair (ℓ, σ) where $\ell \in [s+1]$ and σ is a function $\sigma : V \to \{0,1\}$ that maps every variable of P into a bit in $\{0,1\}$. We define $CONF(P) = [s+1] \times \{\sigma \mid \sigma : V \to \{0,1\}\}$ to be the set of all configurations of P. [11]

If P has n inputs, then for every $x \in \{0,1\}^n$, the *initial configuration of P with input x* is the pair $(0, \sigma_0)$ where $\sigma_0 : V \to \{0,1\}$ is the function defined as $\sigma_0(X_i) = x_i$ for every $i \in [n]$ and $\sigma_0(v) = 0$ for all variables not in X.

[11] Note that $|CONF(P)| = (s+1)2^{|V|}$: can you see why?

An execution of a NAND program can be thought of as simply progressing, line by line, from the initial configuration to the next one:

Definition 3.3 — NAND next step function. For every NAND program $P = (V, X, Y, L)$, the *next step function of P*, denoted by $NEXT_P$, is the function $NEXT_P : CONF(P) \to CONF(P)$ that defined as follows:

For every $(\ell, \sigma) \in CONF(P)$, if $\ell = |L|$ then $NEXT_P(\ell, \sigma) = (\ell, \sigma)$. Otherwise $NEXT_P(\ell, \sigma) = (\ell + 1, \sigma')$ where $\sigma' : V \to \{0,1\}$

is defined as follows:

$$\sigma'(x) = \begin{cases} NAND(\sigma(v),\sigma(w)) & x = u \\ \sigma(x) & \text{otherwise} \end{cases} \tag{3.1}$$

where $(u,v,w) = L_\ell$ is the ℓ-th triple (counting from zero) in L.

For every input $x \in \{0,1\}^n$ and $\ell in[s+1]$, the *ℓ-th configuration of P on input x*, denoted as $conf_\ell(P,x)$ is defined recursively as follows:

$$conf_\ell(P,x) = \begin{cases} (0,\sigma_0) & \ell = 0 \\ NEXT_P(conf_{\ell-1}(P)) & \text{otherwise} \end{cases} \tag{3.2}$$

where $(0,\sigma_0)$ is the initial configuration of P on input x.

We can now finally formally define the notion of computing a function:

Definition 3.4 — Computing a function. Let $P = (V,X,Y,L)$ and $n = |X|$, $m = |Y|$ and $s = |L|$. Let $F : \{0,1\}^n \to \{0,1\}^m$. We say that P *computes* F if for every $x \in \{0,1\}^n$, if $y = F(x)$ then $conf_s(P,x) = (s,\sigma)$ where $\sigma(Y_j) = y_j$ for every $j \in [m]$.

For every $s \in \mathbb{N}$, we define $SIZE(s)$ to be the set of all functions that are computable by a NAND program of at most s lines.

The formal specification of any programming language, no matter how simple, is often cumbersome, and the definitions above are no exception. You should go back and read them and make sure that you understand why they correspond to our informal description of computing a function via NAND programs. From this point on, we will not distinguish between the representation of a NAND program in terms of lines of codes, and its representation as a tuple $P = (V,X,Y,L)$.

Let $XOR_n : \{0,1\}^n \to \{0,1\}$ be the function that maps $x \in \{0,1\}^n$ to $\sum_{i=0}^{n} x_i \pmod 2$. The NAND program we presented above yields a proof of the following theorem

Theorem 3.5 — Computing XOR. $XOR_2 \in SIZE(4)$

Similarly, the addition program we presented shows that $ADD_1 \in SIZE(5)$.

3.3 *Canonical input and output variables*

The specific identifiers for NAND variables (other than the inputs and outputs) do not make any difference in the program's functionality, as long as we give separate variable distinct identifiers. For example, if I replace all instances of the variable `foo` with `boazisgreat` then, under the (unfortunately common) condition that `boazisgreat` was not used in the original program, the resulting program will still compute the same function. For convenience, it is sometimes useful to assume that all variables identifiers have some canonical form such as being either `x_`$\langle i \rangle$, `y_`$\langle j \rangle$ or `work_`$\langle k \rangle$. Similarly, while we allowed in Definition 3.1 the variables to be members of some arbitrary set V, it is sometimes useful to assume that V is simply the set of numbers from 0 to some natural number (which can never be more than three times the number of lines s). This motivates the following definition:

Definition 3.6 — Canonical variables. Let $P = (V, X, Y, L)$ be a NAND program and let $n = |X|$ and $m = |Y|$. We say that P has *canonical variables* if $V = [t]$ for some $t \in \mathbb{N}$, $X = (0, 1, \ldots, n-1)$ and $Y = (t-m, t-m+1, \ldots, t-1)$.

That is, in a canonical form program, the variables are the set $[t]$, where the input variables correspond to the first n variables and the outputs to the last m variables. Every program can be converted to an equivalent program of canonical form:

Theorem 3.7 — Convert to canonical variables. For every $F : \{0,1\}^n \to \{0,1\}^m$ and $s \in \mathbb{N}$, F can be computed by an s-line NAND program if and only if it can be computed by an s-line NAND program with canonical variables.

Proof. The "if" direction is trivial, since a NAND program with canonical variables is just a special case of a NAND program. For the "only if" direction, let $P = (V, X, Y, L)$ be an s-line NAND program computing F, and let $t = |V|$. We define a bijection $\pi : V \to [t]$ as follows: $\pi(X_i) = i$ for all $i \in [n]$, $\pi(Y_j) = t - m + j$ for all $j \in [m]$ and we map the remaining $t - m - n$ elements of V to $\{n, \ldots, t-1-m\}$ in some arbitrary one to one way. (We can do so because the X's and Y's are distinct and disjoint.) Now define $P' = (\pi(V), \pi(X), \pi(Y), \pi(L))$, where by this we mean that we apply π individually to every element of of V,X, Y, and the triples of L. Since (as we leave you to verify) the definition of configurations and computing a function are invariant under bijections of V, P'

computes the same function as P. ■

Given Theorem 3.7, since we only care about the functionality (and size) of programs, and not the labels of variables, we will always be able to assume "without loss of generality" that a given NAND program P has canonical form. A canonical form program P can also be represented as a triple (n, m, L) where n, m are (as usual) the inputs and outputs, and L is the lines. This is because we recover the original representation (V, X, Y, L) by simply setting $X = (0, 1, \ldots, n-1)$, $Y = (t-m, t-m+1, \ldots, t-1)$ and $V = [t]$ where t is one plus the largest number appearing in a triple of L. In the following we will freely move between these two representations. If n, m are known from the context, then a canonical form program can be represented simply by the list of triples L.

Configurations of programs with canonical variables: A *configuration* of a program with canonical variables is a pair (ℓ, σ) where $\sigma : [t] \to \{0,1\}$ and t is the number of variables. We can and will identify such a function σ with a string of t bits. Thus we will often say that a configuration of a canonical program is a pair (ℓ, σ) where $\sigma \in \{0,1\}^t$.

3.4 *Composing functions*

Computing the XOR or addition of two bits is all well and good, but still seems a long way off from even the algorithms we all learned in elementary school, let alone World of Warcraft. We will get to computing more interesting functions, but for starters let us prove the following simple extension of Theorem 3.5

Theorem 3.8 — Computing four bit parity. $XOR_4 \in SIZE(12)$

We can prove Theorem 3.8 by explicitly writing down a 12 line program. But writing NAND programs by hand can get real old real fast. So, we will prove more general results about *composing* functions:

Theorem 3.9 — Sequential composition of functions. If $F : \{0,1\}^n \to \{0,1\}^m$ is a function in $SIZE(L)$ and $G : \{0,1\}^m \to \{0,1\}^k$ is a function in $SIZE(L')$ then $G \circ F$ is in $SIZE(L + L')$, where $G \circ F : \{0,1\}^n \to \{0,1\}^k$ is defined as the function that maps $x \in \{0,1\}^n$ to $G(F(x))$.

Theorem 3.10 — Parallel composition of functions. If $F : \{0,1\}^n \to \{0,1\}^m$ is a function in $SIZE(L)$ and $G : \{0,1\}^{n'} \to \{0,1\}^{m'}$ is a function in $SIZE(L')$ then $F\|G$ is in $SIZE(L + L')$, where $F\|G : \{0,1\}^{n+n'} \to \{0,1\}^{m+m'}$ is defined as the function that maps $x \in \{0,1\}^{n+n'}$ to $F(x_0,\ldots,x_{n-1})G(x_n,\ldots,x_{n+n'-1})$.

P We will prove Theorem 3.9 and Theorem 3.10 using our formal definition of NAND programs. But it is also possible to directly give syntactic transformations of the code of programs computing F and G to programs computing $G \circ F$ and $F \oplus G$ respectively. It is a good exercise for you to pause here and see that you know how to give such a transformation. Try to think how you would write a *program* (in the programming language of your choice) that given two strings `C` and `D` that contain the code of NAND programs for computing F and G, would output a string `E` that contains that code of a NAND program for $G \circ F$ (or $F\|G$).

Before proving Theorem 3.9 and Theorem 3.10, note that they do imply Theorem 3.8. Indeed, it's easy to verify that for every $x \in \{0,1\}^4$,

$$XOR_4(x) = \sum_{i=0}^{3} x_i \pmod 3 = ((x_0 + x_1) \bmod 2) + (x_2 + x_3) \bmod 2) \bmod 2) = XOR_2(XOR_2(x_0,x_1)XOR_2(x_2,x_3)) \tag{3.3}$$

and hence

$$XOR_4 = XOR_2 \circ (XOR_2 \oplus XOR_2) \; . \tag{3.4}$$

Since XOR_2 is in $SIZE(4)$, it follows that $XOR_4 \in SIZE(4 + (4 + 4)) = SIZE(12)$.

Using the same idea we can prove the following more general result:

Theorem 3.11 — Computing parity via NAND programs. For every $n > 1$, $XOR_n \in SIZE(10n)$

We leave proving Theorem 3.11 as Exercise 3.3.

3.5 *Proving the composition theorems*

We now formally prove the "Sequential Composition Theorem" Theorem 3.9, leaving the "parallel composition" as an exercise. The idea behind the proof is that given a program $P = (V, X, Y, L)$ that computes F and a program $P' = (V', X', Y', L')$ that computes G, we can hope to obtain a program P'' that computes $G \circ F$ by simply "copy and pasting" the code for P' after the code for P, replacing the inputs of G with the outputs of F. In our tuple notation this corresponds renaming the variables X' so they are the same as Y, and then making $P'' = (V \cup V', X, Y', L'')$ where L'' is obtained by simply concatenating L and L'.

It is an interesting exercise to try to prove that this transformation works. If you do so, you will find out that you simply can't make the proof go through. It turns out the issue is not about mere "formalities". This transformation is simply not correct: if G and F use the same workspace variable `foo`, then the program P' might assume `foo` is initialized to zero, while the program P might assign `foo` a nonzero value. Thus in the proof we will need to take care of this issue, and ensure that P' and P use disjoint workspace variables. This is one example of a general phenomenon. Trying and failing to prove that a program or algorithm is correct often leads to discovery of bugs in it.

We now turn to the full proof. It is somewhat cumbersome since we have to **(1)** fully specify the transformation of P and P' to P'' and **(2)** prove that the transformed program P'' does actually compute $G \circ F$. Nevertheless, because proofs about computation can be subtle, it is important that you read carefully the proof and make sure you understand every step in it.

Proof of Theorem 6.4. Let $P = (V, X, Y, L)$ and $P' = (V', X', Y', L')$ be the programs for computing F and G respectively, and assume without loss of generality that they are in canonical form, and so $X = [n]$, $Y = X' = [m]$, and $Y' = [k]$. Let $t = |V|$, $s = |L|$, $t' = |V'|$, and $s' = |L'|$. We will construct an $s + s'$ line canonical form program P'' with $t + t' - m$ variables that computes $G \circ F : \{0,1\}^n \to \{0,1\}^k$ (see Fig. 3.5). To specify P'' we only need to define its set of lines $L'' = (L''_0, L''_1, \ldots, L''_{s+s'-1})$. The first s lines of L'' simply equal L. The next s' lines are obtained from L' by adding $t - m$ to every label.[12] In

[12] Since we are representing programs in canonical form, each line is a triple of numbers, and adding $t - m$ simply shifts the set of variables used by P' from $\{0, \ldots, t' - 1\}$ to the last t' elements of $[t + t' - m]$ which is the set of variables of the composed program P''.

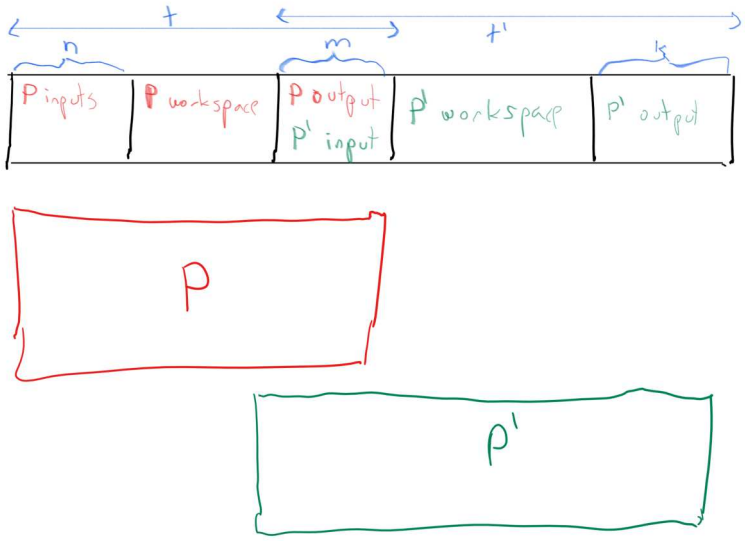

Figure 3.5: Given canonical-variables programs P and P' that compute $F : \{0,1\}^n \to \{0,1\}^m$ and $G : \{0,1\}^m \to \{0,1\}^k$ respectively, we create a program P'' using $t + t' - m$ variables to compute $G \circ F$ where t and t' are the number of variables used by P and P' respectively. In this program, the variables corresponding to the outputs of P and the inputs of P' are shared, but all other variables are disjoint.

other words, for every $\ell \in [s + s']$,

$$L''_\ell = \begin{cases} L_\ell & \ell < s \\ (L'_{\ell-s,0} + t - m, L'_{\ell-s,1} + t - m, L'_{\ell-s,2} + t - m) & \ell > s \end{cases} \tag{3.5}$$

where $L' = ((L'_{0,0}, L'_{0,1}, L'_{0,2}), \cdots, (L'_{t',0}, L'_{t',1}, L'_{t',2})$.

We now need to prove that P'' computes $G \circ F$. We will do so by showing the following two claims:[13]

Claim 1: For every $x \in \{0,1\}^n$ and $\ell \in [s+1]$, $conf_\ell(P'', x) = (\ell, \sigma 0^{t'-m})$ where $(\ell, \sigma) = conf_\ell(P, x)$.[14]

Claim 2: For every $x \in \{0,1\}^n$ and $\ell \in \{s, \ldots, s + s'\}$, $conf_\ell(P'', x) = (\ell, z\sigma)$ where $(\ell - t, \sigma) = conf_{\ell-t}(P', F(x))$ and $z \in \{0,1\}^{t-m}$ is some string.

Claim 2 implies the theorem, since by our definition of P'' computing the function G, it follows that if $(t', \sigma) = conf_{t'}(P'', F(x))$ then the last k bits of σ correspond to $G(F(x))$. We will outline the proof of Claims 1 and 2:

Proof outline of claim 1: The proof follows because the lines $L''_0, \ldots, L''_{t-1}$ are identical to the lines of L, and hence they only touch the first t variables, and leave the remaining $t' - m$ equal to 0.

[13] These two claims are easiest to understand by looking at Fig. 3.5. Claim 1 simply says that in the first s steps of the execution, the state of the first t variables corresponds to the state in the execution of P, and the last $t' - m$ variables are untouched. Claim 2 says that in the following t' step, the state of the first $t - m$ variables remains as they were at the end of the execution of P, and the last t' variables evolve according to the execution of P'. The variables $\{t - m, \ldots, t - 1\}$ are involved in both executions: they play the role of output variables for P and the role of input variables for P'.

[14] Recall that for a program in canonical form, we can think of the state σ as a string, and hence $\sigma 0^{t'-m}$ means that we concatenate $t' - m$ zeroes to this string. Similarly in Claim 2 below, $z\sigma$ refers to the concatenation of the string z to σ.

Proof outline of claim 2: By Claim 1, by step t the configuration has the value $F(x)$ on the variables $t-m,\ldots,t-1$. Since the lines $L''_t,\ldots,L''_{t+t'-1}$ only touch the variables $t-m,\ldots,t+t'-1-m$, the last t' variables correspond to the same configuration as running the program P'' on $F(x)$.

To make these outlines into full proofs, we need to use *induction*, so we can argue that for every ℓ, if we maintained these properties up to step $\ell-1$, then they are maintained in step ℓ as well. We omit the full inductive proof, though working out it for yourself can be an excellent exercise in getting comfortable with such arguments. ■

3.5.1 *Example: Adding two-bit numbers*

Using composition, we can show how to add *two bit* numbers. That is, the function $ADD_2 : \{0,1\}^4 \rightarrow \{0,1\}^3$ that takes two numbers x, x' each between 0 and 3 (each represented with two bits using the binary representation) and outputs their sum, which is a number between 0 and 6 that can be represented using three bits. The grade-school algorithm gives us a way to compute ADD_2 using ADD_1. That is, we can add each digit using ADD_1 and then take care of the carry. That is, if the two input numbers have the form $x_0 + 2x_1$ and $x_2 + 2x_3$, then the output number $y_0 + y_1 2 + y_3 2^2$ can be computed via the following "pseudocode" (see also Fig. 3.6)

```
y_0,c_1 := ADD_1(x_0,x_2) // add least significant digits
z_1,c_2 := ADD_1(x_1,x_3) // add second digits
y_1,c'_2 := ADD_1(z_1,c_1) // second output is sum + carry
y_2 := c_2 OR c'_2 // top digit is 1 if one of the top
    carries is 1
```

To transform this pseudocode into an actual program or circuit, we can use Theorem 3.9 and Theorem 3.10. That is, we first compute $(y_0, c_1, z_1, c_2) = ADD_1 \| ADD_1(x_0, x_2, x_1, x_3)$, which we can do in 10 lines via Theorem 3.10, then apply ADD_1 to (z_1, c_1), and finally use the fact that $OR(a,b) = NAND(NOT(a), NOT(b))$ and $NOT(a) = NAND(a,a)$ to compute `c_2 OR c'_2` via three lines of NAND. The resulting code is the following:

```
// Add a pair of two-bit numbers
// Input: (x_0,x_1) and (x_2,x_3)
// Output: (y_0,y_1,y_2) representing the sum
// x_0 + 2x_1 + x_2 + 2x_3
//
// Operation:
```

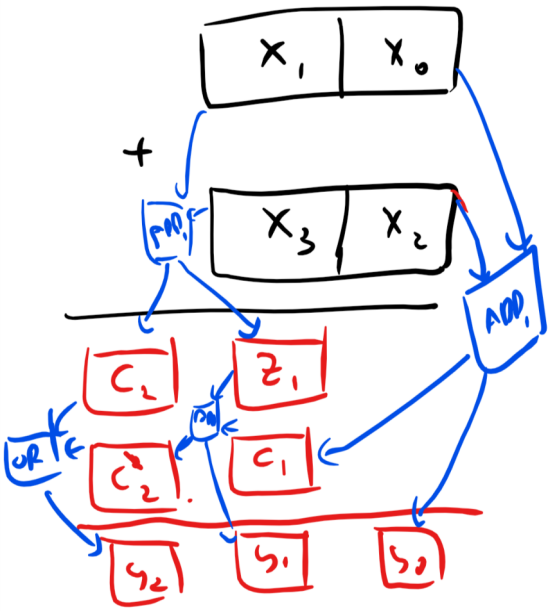

Figure 3.6: Adding two 2-bit numbers via the grade school algorithm.

```
// 1) y_0,c_1 := ADD_1(x_0,x_2):
// add the least significant digits
// c_1 is the "carry"
u := x_0 NAND x_2
v := x_0 NAND u
w := x_2 NAND u
y_0 := v NAND w
c_1 := u NAND u
// 2) z'_1,z_1 := ADD_1(x_1,x_3):
// add second digits
u := x_1 NAND x_3
v := x_1 NAND u
w := x_3 NAND u
z_1 := v NAND w
z'_1 := u NAND u
// 3) Take care of carry:
// 3a) y_1 = XOR(z_1,c_1)
u := z_1 NAND c_1
v := z_1 NAND u
w := c_1 NAND u
y_1 := v NAND w
// 3b) y_2 = z'_1 OR (z_1 AND c_1)
// = NAND(NOT(z'_1), NAND(z_1,c_1))
```

```
u := z'_1 NAND z'_1
v := z_1 NAND c_1
y_2 := u NAND v
```

For example, the computation of the deep fact that $2 + 3 = 5$ corresponds to running this program on the inputs $(0, 1, 1, 1)$ which will result in the following trace:

```
Executing step 1: "u␣␣␣:=␣x_0␣NAND␣x_2" x_0 = 0, x_2 = 1, u
    is assigned 1,
Executing step 2: "v␣␣␣:=␣x_0␣NAND␣u" x_0 = 0, u = 1, v is
    assigned 1,
Executing step 3: "w␣␣␣:=␣x_2␣NAND␣u" x_2 = 1, u = 1, w is
    assigned 0,
Executing step 4: "y_0␣:=␣v␣NAND␣w" v = 1, w = 0, y_0 is
    assigned 1,
Executing step 5: "c_1␣:=␣u␣NAND␣u" u = 1, u = 1, c_1 is
    assigned 0,
Executing step 6: "u␣␣␣:=␣x_1␣NAND␣x_3" x_1 = 1, x_3 = 1, u
    is assigned 0,
Executing step 7: "v␣␣␣:=␣x_1␣NAND␣u" x_1 = 1, u = 0, v is
    assigned 1,
Executing step 8: "w␣␣␣:=␣x_3␣NAND␣u" x_3 = 1, u = 0, w is
    assigned 1,
Executing step 9: "z_1␣:=␣v␣NAND␣w" v = 1, w = 1, z_1 is
    assigned 0,
Executing step 10: "z'_1␣:=␣u␣NAND␣u" u = 0, u = 0, z'_1 is
    assigned 1,
Executing step 11: "u␣␣␣:=␣z_1␣NAND␣c_1" z_1 = 0, c_1 = 0, u
     is assigned 1,
Executing step 12: "v␣␣␣:=␣z_1␣NAND␣u" z_1 = 0, u = 1, v is
    assigned 1,
Executing step 13: "w␣␣␣:=␣c_1␣NAND␣u" c_1 = 0, u = 1, w is
    assigned 1,
Executing step 14: "y_1␣:=␣v␣␣␣NAND␣w" v = 1, w = 1, y_1 is
    assigned 0,
Executing step 15: "u␣␣␣:=␣z'_1␣NAND␣z'_1" z'_1 = 1, z'_1 =
    1, u is assigned 0,
Executing step 16: "v␣␣␣:=␣z_1␣NAND␣c_1" z_1 = 0, c_1 = 0, v
     is assigned 1,
Executing step 17: "y_2␣:=␣u␣NAND␣v" u = 0, v = 1, y_2 is
    assigned 1,
Output is y_0=1, y_1=0, y_2=1
```

3.5.2 *Composition in NAND programs*

We can generalize the above examples to handle not just sequential and parallel but all forms of *composition*. That is, if we have an s line program P that computes the function F, and a program P' that can compute the function G using t standard NAND lines and k calls to a "black box" for computing F, then we can obtain a $t + ks$ line program P'' to compute G (without any "magic boxes") by replacing every call to F in P' with a copy of P (while appropriately renaming the variables).

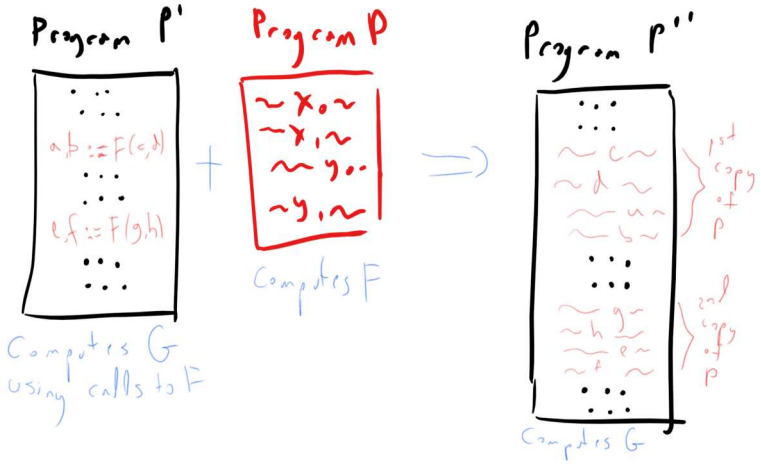

Figure 3.7: We can compose a program P that computes F with a program P' that computes G by making calls to F, to obtain a program P'' that computes G without any calls.

3.6 *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.
- An equivalent formulation is that a function is computable by a NAND program if it can be computed by a NAND circuit.

3.7 *Exercises*

Exercise 3.1 Which of the following statements is false? a. There is a NAND program to add two 4-bit numbers that has at most 100 lines.

b. Every NAND program to add two 4-bit numbers has at most 100 lines.

c. Every NAND program to add two 4-bit numbers has least 5 lines. ■

Exercise 3.2 Write a NAND program that adds two 3-bit numbers. ■

Exercise 3.3 Prove Theorem 3.11.[15] ■

[15] **Hint:** Prove by induction that for every $n > 1$ which is a power of two, $XOR_n \in SIZE(4(n-1))$. Then use this to prove the result for every n.

3.8 Bibliographical notes

The exact notion of "NAND programs" we use is nonstandard, but these are equivalent to standard models in the literature such as *straightline programs* and *Boolean circuits*.

An historical review of calculating machines can be found in Chapter I of the 1946 "operating manual" for the Harvard Mark I computer, written by Lieutenant Grace Murray Hopper and the staff of the Harvard Computation Laboratory.

3.9 Further explorations

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

(to be completed)

3.10 Acknowledgements