projekt: zer0

life to the unliving

Borislav Nikolov

2026-07-14

Table of contents

Machinist and Machine **1**

Preface **3**

1 Symbols **7**

2 Language **37**

3 One Minus One **45**
3.1 Addressible Memory 50
3.2 Subtract 51
3.3 Clock 56
3.4 Computer 57
3.5 Assembly Language 66
3.6 Input and Output 72
3.7 One Word 78
3.8 Dictionary 81
3.9 Jumping Around 82
3.10 Self Modification 91

Machinist and Machine

Since AI can produce language, or at least a sequence of words that look like language, the whole world has spiraled into chaos. Many questions were on the table, what about the jobs, what about ethics, what if its alive, what if it kills us.

The most important question however is ignored: what am I.

What if I am it? What if it is me? Am I alive?

This is a book about the fundamentals of programming and symbolic machines, living and unliving.

Preface

The computational theory of the mind is taken for truth by the general scientific community, besides few heretics and few lunatics, some with nobel prizes, the rest of the world either agrees with it, does not think of it, or is afraid to speak against it.

The Penrose argument has almost fallen, it basically states that our brains can compute uncomputable functions, he tries to find where the super turing tape comes from and tries to find in the quantum properties of the brain. Tegmark computed decoherence happens too fast. Inspite the criticism, there is always missing physics, and it wont be the first time we get surprised.

Now that Large Language Models like ChatGPT are used by everybody for everything, people are even more convinced they are equivalent to artificial neural networks, many prefer talking to language models rather than human beings.

Harari said 'AI is language itself liberating itself from flesh'. Dawkins said 'If I deny Claude consciousness, I deny it to myself'.

I have been using ChatGPT since GPT 2, but after GPT3.5 I used it multiple hours per day, at GPT4 it was almost half my day, and now I use Fable and 5.6Sol 12 hours per day. I love it so much, it is the most amazing machine I have ever spoken with. In the same time I hate it so much, the words that come out of it, when it compliments me, when it pretends to be human. I wish we were not using language to talk to each other.

I will not try to prove anything about consciousness or intelligence, or even discuss those terms. I want to show how arithmetic, language, logic gates, neural networks are all the same thing. The relationship between the expressions and its evolution through time is computation, how the computer computes matters as much as the program it is interpreting and evaluating.

I try to follow the `if` of Maxwell, to understand what is a `bit`, how do you learn it, and more importantly how do you forget it, to show that there is structural difference between how our mind computes and how a silicon computer computes. A `bit` is the ability to distinguish two universes and is observer dependent, forgetting a bit is losing that ability, and is also observer dependent. Here observer does not mean conscious, it means correlated. The energy cost to forget a bit at temperature T is $E_{\text{min}} = k_B T \ln 2$. I say that since the mind is not one thing, "forgetting" a bit is completely different than in sillicon, and I say you can not abstract it in sillicon. Not in the same way you can abstract the ant's behaviour without the ant.

> It is not obvious at a glance why forgetting costs energy but learning does not. To get intuition lets look at it from from classical thermodynamics point of view. The second law of thermodynamics says that entropy of isolated system never decreases. Entropy is measurement of disorder, knowing a piece of information means to somehow interact with it, this increases disorder as its atomic and subatomic jiggling becomes your atomic and subatomic jiggling. Its heat becomes your heat. It could be a tiny tiny amount, and at microscopic level it might even be zero. But the disorder will not decrease. Forgetting is different, you need to throw it out of your system in a way you can no longer remember what it was, so you have to throw disorder outside by making order inside. So, forgetting decreases entropy locally, and you need to do work to do so, which will produce entropy outside.

It is very difficult when you start slicing and dicing the human mind to know which part's outside is which part's inside.

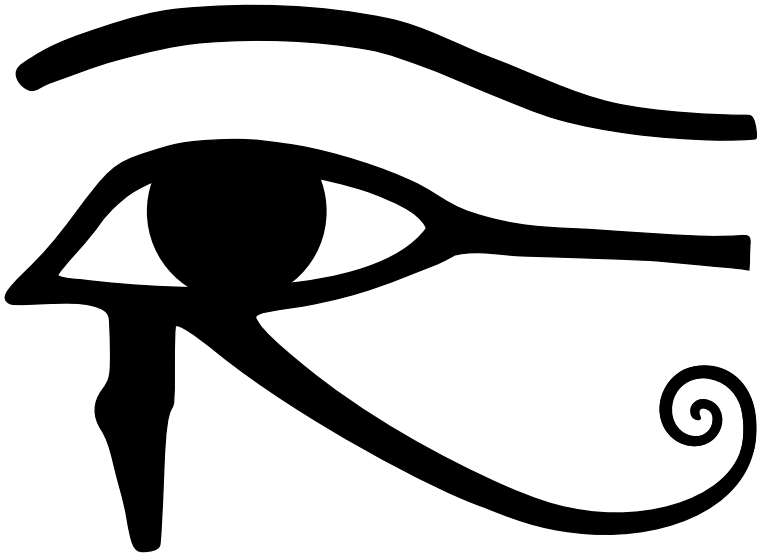
This book is my attempt for understand computers, so that I can find out if I am one.

As Feynman wrote on his last black board: "What I can not create, I do not understand."

I hope you enjoy the book.

PS: As I am writing it, mid way, I already know the answer, and I probably knew it before I began.

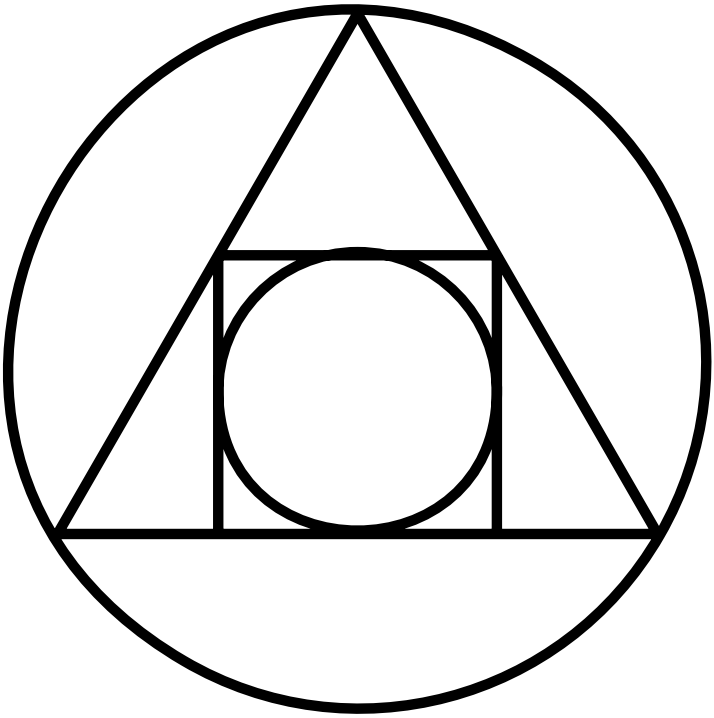
1 Symbols



This is the eye of Horus, the left wedjat eye. He lost it in a battle with Seth. It was restored by Hermes Trismegistus, the Thrice-Great Hermes. It is written in the pyramid of Unas, five thousand years ago:

> Osiris, take the eye of Horus.

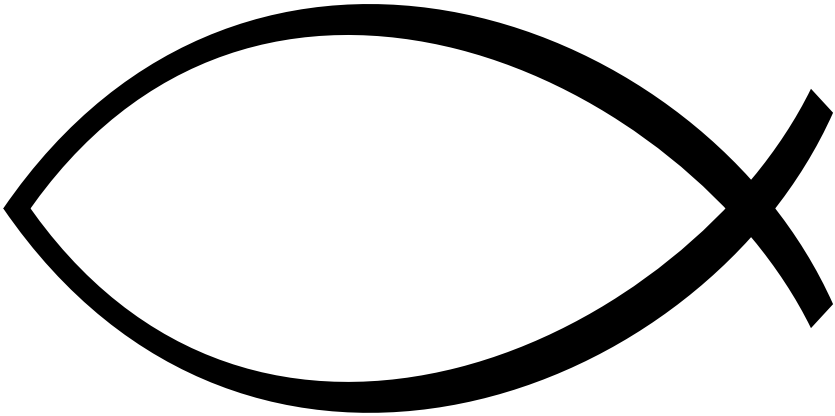
How can a father take his son's eye? Even in death. The son gave everything, asked of nothing.



This is the symbol of the Philosopher's stone, the second Adam, Life out of death.

> One becomes two, two becomes three, and out of the third comes the one as the fourth.

The axiom of Maria Prophetissa, sister of Moses.



Ichthys, ΙΧΘΥΣ, **Ἰ**ησοῦς **Χ**ριστός **Θ**εοῦ **Υ**ἱός **Σ**ωτήρ, Iēsous Christos Theou Huios Sōtēr, or in English: Jesus Christ, Son of God, Savior.

John 3:5:

Greek	Transliteration	English
Ἀμὴν ἀμὴν λέγω σοι, ἐὰν μή τις γεννηθῇ **ἐξ ὕδατος καὶ πνεύματος**, οὐ δύναται εἰσελθεῖν εἰς τὴν βασιλείαν τοῦ θεοῦ	Amēn amēn legō soi, ean mē tis gennēthē **ex hydatos kai pneumatos**, ou dynatai eiselthein eis tēn basileian tou theou	Very truly I tell you, no one can enter the kingdom of God unless they are born **of water and the Spirit**

`ex hydatos kai pneumatos` of water and Spirit, pneumatos is from pneuma, pneuma is living and life giving, wind, breath, spirit.



As above, so below, as within, so without, as the universe, so the soul.

Hermes Trismegistus, Tabula Smaragdina, the Emerald Tablet, Secret of Creation, and it continues, "All things were from One, Its father is the Sun and its mother is the Moon."

What does it mean? How can a circle be squared? How can one be born again of water and wind? How can one come out of the third? How can Miriam become Maria, and Thoth become Hermes, they lived thousands of years apart? How can anything have any meaning?

We are shaped by violence. For me to live, something must die, for me to see, light must crash into my eyes, for me to stand, the floor must push me. Every single experience is mine, and mine alone. I see the world inside out, everything I see, is made in me and from me. The sun, the sound of water, the taste of a good steak, the smell of fresh bread. How then, can we understand each other through this text? My eye is not your eye, my sun is not your sun.

Symbolic language is mankind's greatest creation. It is how two humans can change each other. It is what allows for God to change men, and men to change God. It is what gives time and space to culture. The symbols that come out of us are the air the Geist breathes, and the water it drinks.

Terry Pratchett said, always start at the beginning, it is the best place to start.

> In the beginning was the Word, and the Word was with God, and the Word was God. – John 1:1

Once a word comes out of my mind and gets written in this page, it is as if I have collapsed everything I am into a dot. For practicality, lets say I write the word 'God'. It itself is composed of symbols 'G', 'o', 'd', each of them has their own story, the letter 'd' for example comes from the Egyptian symbol for door or fish, now each of those letters by themselves are nothing, rearranged dog - quite different than God, gdo, dgo and odg - nonsense, God however, has gravity (or maybe is gravity, we are not sure). I guarantee you, when I write this word, it is **nothing**, like what you will read. Not in the same way as 'my sun is not your sun', if I ask you to point to the sun on a hot day, I know where you will point, but if I ask you to point at your God, even if we believe in the same God, we can spend a life discussing and in the end we will be further apart than before. How is that possible?

A symbol exists only in the eye of the beholder, its interpreter and evaluator. It has no meaning on its own. For example, if lightning strikes a tree, and it so happens that it carves the word 'God' on it, if a parrot sees it happen in front of their eyes, it wont matter at all, if you see it, I bet you will freak out.

To interpret a symbol means to relate it to other symbols. For example, `2 + 3` you might think 2 is just related to + and 3, but when you start pulling the string, you will bring all of humanity with it, everything is connected to everything else. The concept of 'twoness', two is human, what is three, but the 'birth of a child'. What is plus but to consume, as three consumes two, heaven is born.

This is what it means to do symbolic interpretation, it is to resolve the relationship between the symbols. This is what symbols 'mean', their meaning is encoded in the way they are related to everything else.

Does the encoding of the symbol matter? `God` or `θεός`, `71 111 100`, `01000111 01101111 01100100`. Not for the symbol itself, but it matters for the interpreter, so it can look up its meaning.

To evaluate means to 'execute' the symbolic expression, for example when you read `1 + 1`, you instantly think of `2`. Try it, try reading `1 + 1` and not think of `2`, I dare you. One issue with this explanation is that the evaluator lives within the symbol space. As in, the act of evaluation can change the evaluator themselves.

This is the act of computation, the evaluation and interpretation of symbols.

Human beings can perform computation, and it seems to be deeply connected with the way we think and reason. This does not make us computers, in the same way that our hearts do not make us pumps, even though they are pumping 5 liters per minute non-stop for a hundred years. Just like all fingers are digits, but not all digits are fingers, all humans can compute, but not all computers can human.

Now, think for a second, can you interpret an expression without evaluating it? and can you evaluate without interpreting it?

DUM DUM DUUUUM

The only thing missing from our story is `choice`, how do we know if we should do one or the other? `1 + 1`, how do you know you are done

evaluating each symbol and know this is its final form? You can spend your whole life to understand what it means, and some monks do, some are satisfied with: it means the number one added to the number one which is two. But how do they know to stop?

Choice is the main character of this book, or any human book for that matter. By the end of the book I hope to convince you, and myself, that the choice of men is different than the choice of machine.

Imagine an aquarium, inside there are trillions of trillions of gas particles, nitrogen, oxygen, argon.. a whole spectacle, air as we call it, we split it in two, with a piece of paper, and we open a hole, so tiny, only a few molecules wide, and we make a demon to guard it, if a particle comes from the left side and its velocity is higher than some value, it will close the hole. If it comes from the right and velocity is below the threshold it will open the gate. This way one part of the aquarium will be very hot, and the other will be very cold. Why is it then, in the heat waves of Europe, we suffer? Why cant we employ those demons to not let hot particles in our houses?

To answer that first you must understand what does it mean to `choose`. The minimal choice is to pick a path based on a condition, if the condition is true, pick the left path if the condition is false pick the right path. It might seem that the condition itself is chosen already, as in we know to put `true` or `false` in it ahead of time, but when you add time and loops you can encode memory and choice with just selects. Let me show you.

```
if true  then A else B -> A
if false then A else B -> B
```

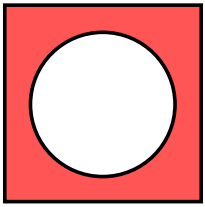
It is called selection, `select(C, A, B)` where `C` is `true` or `false`, one bit of information, 1 or 0, `select(1, A, B)` produces `A`, and `select(0, A, B)` produces `B`.

Select can discard any values, e.g. `select(1, 5, 7)` → `5` doesnt really care what `a` and `b` are, `select(1, APPLE, ORANGE)` → `APPLE`, select just selects conditionally based on the first value `C`.

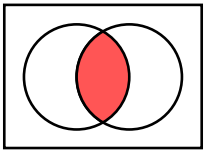
If we reduce `A` and `B` to binary, `TRUE/FALSE` `1/0`, we can create the following logic operators:

```
NOT(a)    = select(a, 0, 1)
AND(a, b) = select(a, b, 0)
OR(a, b)  = select(a, 1, b)
XOR(a,b)  = select(a, NOT(b), b) = select(a, select(b,0, 1), b)
NAND(a,b) = NOT(AND(a,b)) = select(select(a,b,0), 0, 1)
```

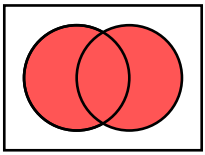
`NOT` is negation, `NOT(true)` is `false`, `NOT(false)` is `true`.



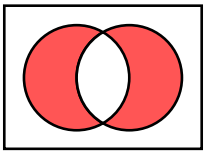
`AND` is intersection, both inputs must be true.



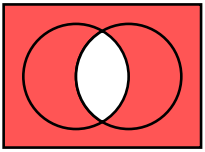
`OR`, is union, at least one must be true.



`XOR`, exclusive `OR`, either one, but not both nor none.



`NAND`, `NOT AND`, alternative denial, it can be either one or none, but not both.



Lets play them out:

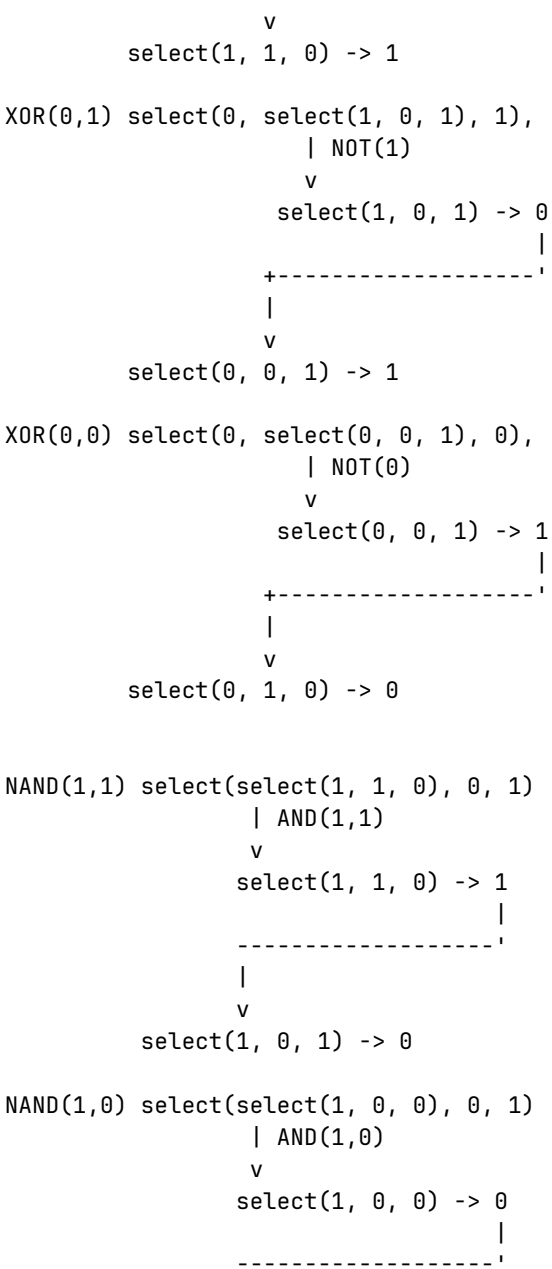
```
NOT(1) select(1, 0, 1) -> 0
NOT(0) select(0, 0, 1) -> 1

AND(1,1) select(1, 1, 0) -> 1
AND(1,0) select(1, 0, 0) -> 0
AND(0,1) select(0, 1, 0) -> 0
AND(0,0) select(0, 0, 0) -> 0

OR(1,1) select(1, 1, 1) -> 1
OR(1,0) select(1, 1, 0) -> 1
OR(0,1) select(0, 1, 1) -> 1
OR(0,0) select(0, 0, 0) -> 0

* we will expand NOT(b) into select(b, 0, 1)
* evaluate it, and then use it in the enclosing select
XOR(1,1) select(1, select(1, 0, 1), 1),
                      | NOT(1)
                      v
                    select(1, 0, 1) -> 0
                                       |
                   +-------------------'
                   |
                   v
         select(1, 0, 1) -> 0

XOR(1,0) select(1, select(0, 0, 1), 0),
                      | NOT(0)
                      v
                    select(0, 0, 1) -> 1
                                       |
                   +-------------------'
                   |
```


v
select(1, 1, 0) -> 1
XOR(0,1) select(0, select(1, 0, 1), 1),
| NOT(1)
v
select(1, 0, 1) -> 0
|
+--------------------'
|
v
select(0, 0, 1) -> 1
XOR(0,0) select(0, select(0, 0, 1), 0),
| NOT(0)
v
select(0, 0, 1) -> 1
|
+--------------------'
|
v
select(0, 1, 0) -> 0
NAND(1,1) select(select(1, 1, 0), 0, 1)
| AND(1,1)
v
select(1, 1, 0) -> 1
|
--------------------'
|
v
select(1, 0, 1) -> 0
NAND(1,0) select(select(1, 0, 0), 0, 1)
| AND(1,0)
v
select(1, 0, 0) -> 0
|
--------------------'

```
                |
                v
          select(0, 0, 1) -> 1

NAND(0,1) select(select(0, 1, 0), 0, 1)
                 | AND(0,1)
                 v
                select(0, 1, 0) -> 0
                                   |
                -------------------'
                |
                v
          select(0, 0, 1) -> 1

NAND(0,0) select(select(0, 0, 0), 0, 1)
                 | AND(0,0)
                 v
                select(0, 0, 0) -> 0
                                   |
                -------------------'
                |
                v
          select(0, 0, 1) -> 1
```

a	NOT
1	0
0	1

a	b	AND	OR	XOR	NAND
1	1	1	1	0	0
1	0	0	1	1	1
0	1	0	1	1	1
0	0	0	0	0	1

Now, lets make a useful choice machine, we will make a machine that can add two numbers. Since we can do only boolean operations, we must encode our number in a sequence of TRUE/FALSE, 1/0, binary.

```
0        F     0
```

```
1        T     1
2       TF    10
3       TT    11
4      TFF   100
5      TFT   101
6      TTF   110
7      TTT   111
8     TFFF  1000
9     TFFT  1001
10    TFTF  1010
11    TFTT  1011
12    TTFF  1100
13    TTFT  1101
14    TTTF  1110
15    TTTT  1111
```

Binary addition works the same as any addition, you do it column by column and carry if you have to carry, remember how you used to do addition before you had a calculator in your pocket:

```
            1           1           1           1
  4782     4782        4782        4782        4782
+ 5739   + 5739      + 5739      + 5739      + 5739
------   ------      ------      ------      ------
             1          21         521       10521
         2 + 9 = 11  1+8+3 = 12  1+7+7 = 15  1+4+5 = 10
         we carry 1  we carry 1  we carry 1  we put 0, and carry
                                             becomes leading 1
```

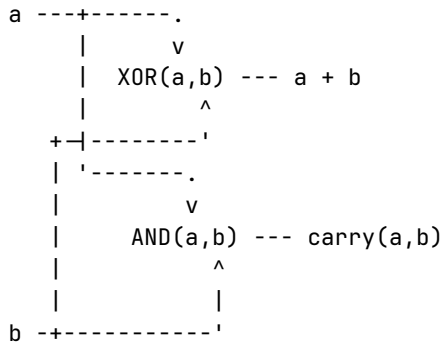
Lets add 6 + 7 in binary:

```
                        1                       1
   110      110         110                     110
 + 111    + 111       + 111                   + 111
   ---      ---         ---                     ---
            1           01                     1101
         0 + 1 = 1   1 + 1 = 2              1+1+1 = 3
                     2 = 10                 3 = 11
                     we put 0 and carry 1   we put 1, and carry
                                            becomes leading 1
```

First how do we add two bits, and we need to know if we have to carry.

```
a b   a+b   carry
1 1   0     1
1 0   1     0
0 1   1     0
0 0   0     0
```

If you look at the output table of our logic machines you see that `SUM(a,b)` is exactly `XOR(a,b)`, and `CARRY(a,b)` is `AND(a,b)`.



This logic circuit is called 'half adder', however to be able to add more than one bit we need to know the previous carry, so we actually need a circuit to add 3 bits, a,b,carry.

```
half adder (a,b):
  HSUM(a,b)   = XOR(a,b)
  HCARRY(a,b) = AND(a,b)

full adder (a,b,c):
  FSUM(a,b,c)   = XOR(XOR(a,b), c)
                  SUM(SUM(a,b), c)
                  a b c   a+b+c
                  1 1 1   1
                  1 1 0   0
```

```
                  1 0 1   0
                  1 0 0   1
                  0 1 1   0
                  0 1 0   1
                  0 0 1   1
                  0 0 0   0

  FCARRY(a,b,c) = OR(AND(a,b), AND(c, XOR(a,b)))

                  we either carry from adding a + b, or
                  we carry from the previous carry if a + b is 1

                  OR(CARRY(a,b),
                     CARRY(c, SUM(a,b)))

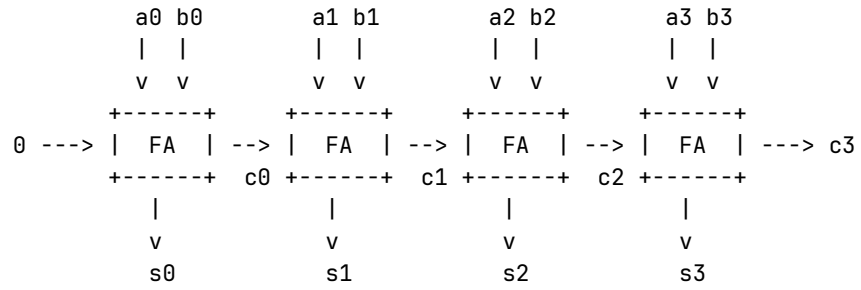
                  a b c   carry
                  1 1 1   1
                  1 1 0   1
                  1 0 1   1
                  1 0 0   0
                  0 1 1   1
                  0 1 0   0
                  0 0 1   0
                  0 0 0   0
```

Lets join 4 full adders:

```
    a3 a2 a1 a0
    b3 b2 b1 b0
  + -----------
 c3 s3 s2 s1 s0

we start with input carry 0:

s0 = FSUM(a0, b0, 0)     c0 = FCARRY(a0, b0, 0)
s1 = FSUM(a1, b1, c0)    c1 = FCARRY(a1, b1, c0)
s2 = FSUM(a2, b2, c1)    c2 = FCARRY(a2, b2, c1)
s3 = FSUM(a3, b3, c2)    c3 = FCARRY(a3, b3, c2)
```




```
Each box is the same
          carry_in    carry_out
FA: (a, b, cin) -> (s, cout)

s = FSUM(a, b, cin) = XOR(XOR(a,b), cin)

    x = XOR(a,b)   = select(a, select(b,0,1), b)
    s = XOR(x,cin) = select(x, select(cin,0,1), cin)

    s = select(select(a, select(b,0,1), b),      # x = XOR(a,b)
               select(cin, 0, 1),                # NOT(cin)
               cin)


cout = FCARRY(a, b, cin) = OR(AND(a,b), AND(cin, XOR(a,b)))

cout = select(select(a, b, 0),                  # AND(a,b)
              1,                                # OR: if set, carry
              select(cin,                       # AND(cin, x)
                     select(a,
                            select(b,0,1),
                            b),                 # x = XOR(a,b)
                     0))
```

Try it on a piece of paper, just add `2 + 1`.



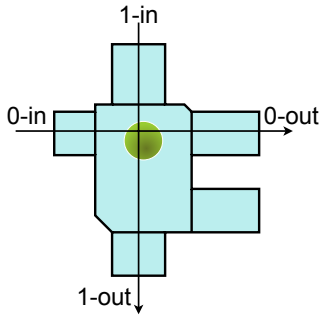
```
          |  |          |  |
          v  v          v  v
        +------+      +------+
 0 ---> |  FA  | --> |  FA  | -->
        +------+  c0 +------+  c1
           |            |
           v            v
           s0           s1

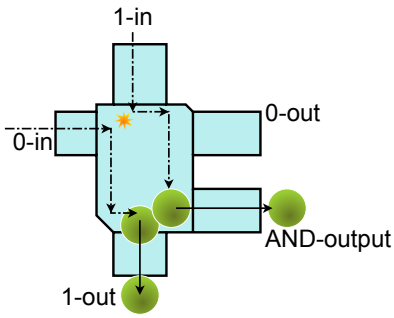
   1  0
   0  1
 + -----
c1 s1 s0
```

```
s0 = XOR(XOR(0,1), 0)
c0 = OR(AND(0,1), AND(0, XOR(0,1)))

s1 = XOR(XOR(1,0), c0)
c1 = OR(AND(1,0), AND(c0, XOR(1,0)))
```

Amazing right? You can simulate this primitive logical circuit in your head, pretend you are it, execute the selections and produce result just as it would if it was made out of wires, water, gravity or billiard balls for that matter.

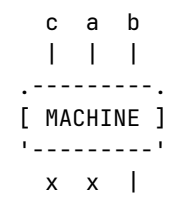




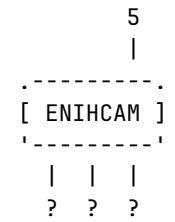
There are many ways to do a `select(c,a,b)` → `y`, but the structural question we must answer is what do we do with `c` the path we want to

select, and with `a` or `b`, whichever one we did not select, and how do we physically do the choice itself.

One option is we have to destroy them.



First, ask yourself what is information, how do you make it and how do you delete it. It is a practical physical question. One bit of information is the ability to distinguish between two universes. DUM DUM DUUUUM. There is no such thing as creating information, or destroying information, we are just moving information from one place to another. Information is distinguishability. The question is 'ability to distinguish' for WHO? One man's trash is another man's treasure. So, in order for us to destroy information we must make it noise from our point of view, we must "forget" about it, meaning it should not be possible for us to go back in time and reconstruct it. It is quite clear in this machine if `select(1,5,7)` $\rightarrow$ `5` and `select(1,5,8)` $\rightarrow$ `5`, we have no idea what produced 5, did it come from `a` or `b`, nor do we know what `c` was. You can see it clearly when we invert the machine:



Our universe does not forget, you cant just make noise, to make noise means the distinction of the states must go somewhere outside of your system.

It is actually computed by Landauer what is the minimum amount of energy required to "forget" one bit of information, such that relative to the observer it becomes noise (undistinguishable), Landauer's limit:

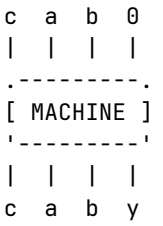
$$E_{\text{min}} = k_B T \ln 2$$

k_B is Boltzmann's constant $k_B = 1.380649 \times 10^{-23}\ \frac{\text{J}}{\text{K}}$. At room temperature ($T \approx 300$ K):

$$k_B T \ln 2 \approx 1.38 \times 10^{-23} \times 300 \times 0.693 \approx 2.9 \times 10^{-21}\ \text{J}$$

So with the energy of 1 green photon of light we can erase about 100 bits. Again I am using the word `erase` but nothing is erased.

There are other ways to actually do the select. We can also pre-make the information channel and modify it.

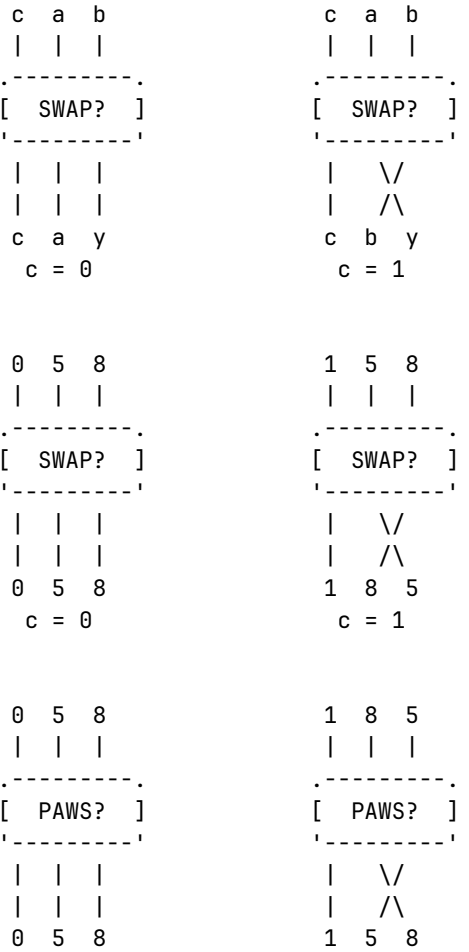


Works backwards in time too.



Lets say we have a mechanism to do `(c, a, b, 0 XOR y) = (c, a, b, y)`, now it seems we dont need any energy for `y` because we just take 0 and write `y`, but sadly, to distinguish between `0` and `y` we need to know that `0` is actually `0`, so somehow we need to have a supply of zeroes. Which again costs energy. And worst of all we need to somehow keep `c,a,b` until the end of our computation and then decide what to do.

There is another way, we do conditional swap.



```
c = 0                    c = 1
```

Swap is completely reversible, but, you must encode information in the manifold, as in your whole design and the way you compose the swap machines must know that the second channel is the result of the select. And you need to figure out what you do with the other signals.

We have a choice to trade heat for time and space, we can produce noise at each step, or we can keep the garbage bouncing around by designing a very complicated board where the computation happens, until the computation is done and then we can release heat at once, or somehow unwind the computation backwards.

This is a very active area of research, some machines can compute the same result with 1/10000th of the energy needed.

But you can see now why our demon can not help us, it needs to `select`, and select produces heat, either now, before or later.

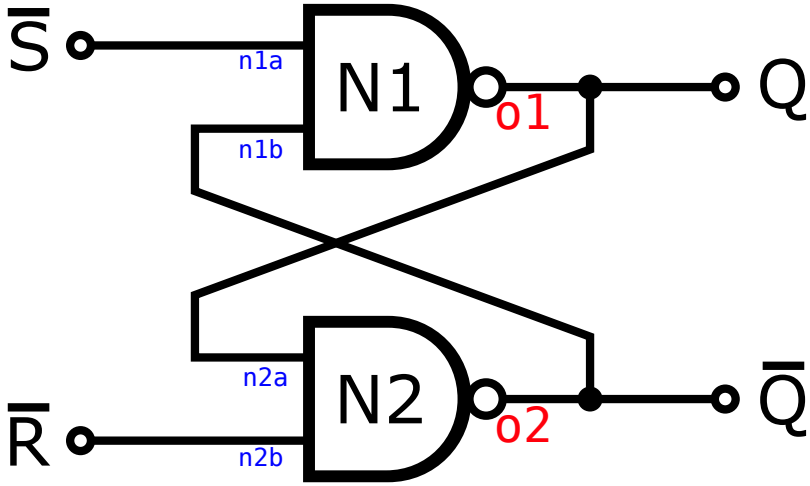
So, what about the demons in your head? How are you doing the selection? How are you erasing the choice? Lets read it again: `1 + 1 = ?`, 2 appears in your mind. Where do we even begin? How do you see? How can you distinguish shapes? How can you look up and understand `1` and discriminate the concept of `oneness` from the number `1`? How do you automatically do the addition without thinking? At the same time random thoughts about `XOR` `AND` and `select(c,a,b)` are rushing around in your mind, how do you even think?



Is there a difference between `select(c,a,b)` and the choice Abraham made to sacrifice his only son? Or when Horus gave his eye to resurrect Osiris? Or if an evaluator would evaluate or interpret a symbol? How can it square a circle? How many bits are needed to encode God? Can we `select(1, Nothing, God)`, how much energy is needed to erase God?

Again, a symbol is nothing on its own, we decide that TTTT is 15, we can decide its `P` as its the 15th letter of the alphabet, `select(1, TTTT, 0)` could be `select(1, 15, 0)` or `select(1, P, A)` it is entirely up to us, and our `select` machines to decide what we do with the bits, and what kind of circuits we build with them.

Now, lets talk for a second where do the paths live, `a` and `b`. We will capture one bit of information in a loop of selects:



The round things in the middle N1 N2 are NAND gates, $\bar{S}$ and $\bar{R}$ are the inverse of S and R, e.g. if S is 0, $\bar{S}$ is 1, thats what the bars on top mean, inverse. Now, lets write it down in selects.

```
NAND(a,b) = NOT(AND(a,b)) = select(select(a,b,0), 0, 1)

input S̄
input R̄
output Q̄
output Q

n1a = S̄
n2b = R̄

o1 = n2a = Q
o2 = n1b = Q̄

N1 = NOT(AND(n1a, n1b)) -> o1
N2 = NOT(AND(n2a, n2b)) -> o2
```

```
Since:
  o2 and n1b are the same signal
  o1 and n2a are the same signal
We can rewrite it as:

N1 = NOT(AND(n1a, o2)) -> o1
N2 = NOT(AND(o1, n2b)) -> o2

Or compiled to selects:

o1 = select(select(n1a, o2, 0), 0, 1)
o2 = select(select(o1, n2b, 0), 0, 1)

Q = select(select(S̄, Q̄, 0), 0, 1)
Q̄ = select(select(Q, R̄, 0), 0, 1)
```

We are in a bit of a pickle, because Q is defined in terms of $\bar{Q}$ and $\bar{Q}$ in terms of Q, in an infinite recursion, but the physical world is not symbolic, it evolves in time and all at once, so the proper way to define the expression in our physical world is: at time t+1 Q uses the value of $\bar{Q}$ at time t, and at t+1 $\bar{Q}$ uses Q at time t.

```
Q(t+1) = NAND(S̄, Q̄(t))
Q̄(t+1) = NAND(Q(t), R̄)

Q(t+1) = select(select(S̄, Q̄(t), 0), 0, 1)
Q̄(t+1) = select(select(Q(t), R̄, 0), 0, 1)
```

Lets run it step by step:

```
Phase 1: WRITE 1                          S̄ = 0, R̄ = 1
-------------
Q(t+1) = select(select(0, Q̄(t), 0), 0, 1)
Q̄(t+1) = select(select(Q(t), 1, 0), 0, 1)

t0: Q = ?
    Q̄ = ?                                 born unknown
```

```
t1: Q = select(select(0, ?, 0), 0, 1)     reads Q̄(t0)=?
              select(0, ?, 0) = 0         c=0, pick b=0
                                          (? discarded, unread)
        select(0, 0, 1) = 1               c=0, pick b=1
    Q = 1
    Q̄ = select(select(?, 1, 0), 0, 1)     reads Q(t0)=?
              select(?, 1, 0) = ?         c unknown, could pick 1 or 0
        select(?, 0, 1) = ?
    Q̄ = ?

t2: Q = select(select(0, ?, 0), 0, 1)     reads Q̄(t1)=?
              select(0, ?, 0) = 0         ? discarded again
        select(0, 0, 1) = 1
    Q = 1
    Q̄ = select(select(1, 1, 0), 0, 1)     reads Q(t1)=1
              select(1, 1, 0) = 1         c=1, pick a=1
        select(1, 0, 1) = 0               c=1, pick a=0
    Q̄ = 0                                 the last ? is dead

t3: Q = select(select(0, 0, 0), 0, 1)     reads Q̄(t2)=0
              select(0, 0, 0) = 0
       select(0, 0, 1) = 1
    Q = 1
    Q̄ = select(select(1, 1, 0), 0, 1)     reads Q(t2)=1
              select(1, 1, 0) = 1
        select(1, 0, 1) = 0
    Q̄ = 0                                 settled: (1, 0)

--- inputs change: S̄ = 1, R̄ = 1 ---

Phase 2: HOLD / READ
-------------
Q(t+1) = select(select(1, Q̄(t), 0), 0, 1)
Q̄(t+1) = select(select(Q(t), 1, 0), 0, 1)

t4: Q = select(select(1, 0, 0), 0, 1)     reads Q̄(t3)=0
              select(1, 0, 0) = 0         c=1, pick a=0
        select(0, 0, 1) = 1
    Q = 1
    Q̄ = select(select(1, 1, 0), 0, 1)     reads Q(t3)=1
```

```
                select(1, 1, 0) = 1
            select(1, 0, 1) = 0
        Q̄ = 0

t5: Q = select(select(1, 0, 0), 0, 1) = 1 reads Q̄(t4)=0
    Q̄ = select(select(1, 1, 0), 0, 1) = 0 reads Q(t4)=1

    * READ: Q = 1 nobody asserting, the loop feeds itself

--- inputs change: S̄ = 1, R̄ = 0 ---

Phase 3: WRITE 0
-------------
Q(t+1) = select(select(1, Q̄(t), 0), 0, 1)
Q̄(t+1) = select(select(Q(t), 0, 0), 0, 1) <- R̄ slot recompiled to 0

t6: Q = select(select(1, 0, 0), 0, 1)   reads Q̄(t5)=0
                select(1, 0, 0) = 0
            select(0, 0, 1) = 1
        Q = 1                           old value, one last tick
        Q̄ = select(select(1, 0, 0), 0, 1)   reads Q(t5)=1
                select(1, 0, 0) = 0     c=1, pick a=0: stored 1
                                        merged away
            select(0, 0, 1) = 1
        Q̄ = 1                             forced; transient (1,1)

t7: Q = select(select(1, 1, 0), 0, 1)     reads Q̄(t6)=1
                                          the news arrives
                select(1, 1, 0) = 1
            select(1, 0, 1) = 0
        Q = 0                             <- flipped
        Q̄ = select(select(1, 0, 0), 0, 1)     reads Q(t6)=1
                select(1, 0, 0) = 0
            select(0, 0, 1) = 1
        Q̄ = 1

t8: Q = select(select(1, 1, 0), 0, 1) = 0 reads Q̄(t7)=1
    Q̄ = select(select(0, 0, 0), 0, 1) = 1 reads Q(t7)=0
    settled: (0, 1)
```

```
--- inputs change: S̄ = 1, R̄ = 1 ---

Phase 4: HOLD / READ
-------------
Q(t+1) = select(select(1, Q̄(t), 0), 0, 1)
Q̄(t+1) = select(select(Q(t), 1, 0), 0, 1)

t9: Q = select(select(1, 1, 0), 0, 1)     reads Q̄(t8)=1
               select(1, 1, 0) = 1
        select(1, 0, 1) = 0
    Q = 0
    Q̄ = select(select(0, 1, 0), 0, 1)     reads Q(t8)=0
               select(0, 1, 0) = 0        c=0, pick b=0
       select(0, 0, 1) = 1
    Q̄ = 1

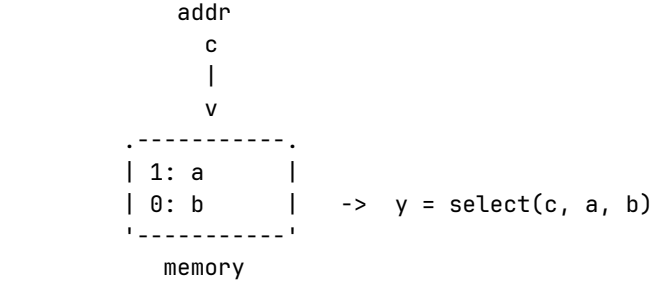
t10: Q = select(select(1, 1, 0), 0, 1) = 0 reads Q̄(t9)=1
     Q̄ = select(select(0, 1, 0), 0, 1) = 1 reads Q(t9)=0

     * READ: Q = 0 same inputs as t5, different answer
```

Pause for a moment to think what is actually happening.

We are storing 1 bit of information in the time evolution of the infinite loop between Q and Q̄. By having an infinite loop of selects we can now have memory. How weird is that?

Even weirder, not only can we have memory from infinite `choice` loops but we can have `choice` from the value at specific memory location at specific time:



So, time gives us memory, and moments (slices through time) give us choice.

Notice how we have not spoken about what kind of mechanisms we use to do the `select`, could be gravity, could be electricity, could be water and valves, or brain cells, anything that can choose between two paths will work. There are various practical costs of course, how fast it is to do the select, how much energy is lost, when do we have to pay the information loss price and so on, but those are just technicalities. Later in the book we will discuss specifically how modern transistors do select, but for now, we focus on the `select` as a primitive.

`select(c,a,b)` → `y` has 3 signals coming in and one signal going out. The act of a machine taking inputs and producing an output is called `application`, I did not explain it as a thing, and you were doing it in your head. It is recursive all the way down. If you look at `NAND: NOT(AND(a,b))`, first we must `apply AND` to `a` and `b` and then we `apply NOT` to the result of `AND`, whatever it is. You can not `apply NOT` until `AND` is done.

Imagine this expression

```
NOT( AND( NOT( XOR(a,b) ), d ) ) -> x

     XOR(a,b)  -> p   innermost first
     NOT(p)    -> q
     AND(q, d) -> r
     NOT(r)    -> x   something comes out
```

Naturally we have to start applying from inside out. First `XOR(a,b)`, then the `NOT`, then the `AND`, then the most outside `NOT`. Deep deep down, select actually does not care what `a` and `b` are, it only cares what `c` is. Not even what it is, but does it pass the actual physical threshold that makes it choose between `a` and `b`, does it have enough momentum, or voltage, or weight, whatever the underlying system needs to do a swap, or suppress one of the inputs. `AND(a, b) = select(a, b, 0)` if a is 0, the answer is 0, `b` was never applied, `b` could be an unfinished cascade of a thousand selects. It could be a cascade that never finishes. We could build our recursive application machine in a way that it will never know what `b` could be, or we can make it so it first realizes `b` and then ignores it. It is our choice as machine designers.

If we make two different machines, one which applies all selects recursively, and the other one only applies the needed selects, do you think we can distinguish between them?

The answer is yes, imagine `AND(a, b)` where `b` is an infinite loop of selects, and `a` does not pass the threshold, the machine that has to apply the circuit of selects at `b` will be stuck, while the other machine will produce 0.

What does `forgetting` or `suppressing` mean when we never realize `b`? Unrealized `b` never created a distinguishable universe, we do not need to pay a price for creating an undistinguishable universe. Meaning the information of `b` was never "created" and we do not need to "erase" it.

Another good question to ask is, how do you know the machine is stuck? Maybe `b`'s circuit is so complicated it takes 1 year to complete, is it stuck? We know that one machine produced a result, we do not know if the other is stuck or busy.

Imagine that our universe is executed on a symbolic machine. At which point can Abraham choose between God and Isaac? `select(c, God, Isaac)` $\rightarrow$ `?` there are two questions, first, at which point `c` is chosen, and more importantly by what circuit was it chosen, and second, is the suppressed input realized, either God or Isaac. is `c` chosen by Abraham's circuitry, or by the machine that evaluates it? Or are they the same thing? and Abraham is just a circuit part of a bigger circuit, in which case, can we even say that he made a choice?

When you zoom in there is no difference between any part of the machine, everything is just selects, latches or adders, demons or people, there is no difference between the outer machine and inner machines. It is the same machine, you could say that Abraham is an inner machine of the outer machine, but you can point at any machine and rotate the whole circuit around it. It is arbitrary where you draw the line between machines, and depends on what you are looking at. Maybe you look at the earth as one machine, maybe you look at a cell as one machine, maybe the whole universe.

> As above, so below, as within so without, as the universe so the soul.

I have hand-waved over quite some philosophical, thermodynamical, and theological arguments, and many of the things I have said are under fierce debate from within those communities. For now all is well, lets try to actually build more complicated systems with selects and then we will see where that leads us.

The one point I wish you would keep from this chapter: symbols, their relationships and how we choose between them are the very core of any computation, their encoding is not important.

You can make an ant colony without ants.

2 Language

What is a language? A structured system of communication that consists of grammar and vocabulary. English is a language, it has grammar: rules of what sequences are allowed, and vocabulary, a graph of relationships between words.

For example:

```
sentence: it is raining

it: used as the subject or object of a verb or after a preposition
is: verb, be
raining: verb, to rain
rain: water that falls from the sky in separate drops
water: a liquid without colour, smell or taste that falls as rain,
       is in lakes, rivers and seas, and is used for drinking,
       washing, etc.
...
```

You can see how each word is related to each other, and some definitions are somewhat circular, for example rain defines water, and water defines rain.

The English language is open, there are grammar rules how to create new words, and if enough people use the new word it becomes part of the language. Think for a second how new words appear, a bunch of people extend the language, often even just a baby misuses a word, like saying pasketti instead of spaghetti, and their family starts calling it pasketti, then it grows, someone opens a shop, in the menu they put pasketti because they think its cute, it becomes viral via social media, in a month everyone is calling spaghetti pasketti. But for a while for most people pasketti is just a nonsense word, they dont even know what it is supposed to mean. Or even worse, for some it means spaghetti, for

some other tribe it means a skateboard trick. And now it is completely ambiguous until one tribe wins. In fact most words have an ambiguous definition, for example `rain`, what if the drops are frozen? Then its snow, but the definition of rain does not rule out snow.

The grammar rules are also somewhat loose, and there are contradictions in them, and most problematically, they are written in the language they describe. Imagine the following rule:

> Each sentence must finish with a full stop, a question mark or an exclamation mark.

So, how can we read sentences before we understand how sentences end? But we must first describe what the word 'sentence' means, which is a sentence.

> sentence: A set of words expressing a statement, a question or an order, usually containing a subject and a verb. In written English sentences begin with a capital letter and end with a full stop/period (.), a question mark (?) or an exclamation mark (!).

Somehow we must evaluate and interpret the language at the same time, and all the rules have to be applied while we are reading the vocabulary, but we must read the vocabulary to understand how to even read the rules. And the rules change often, and the vocabulary changes from place to place or from time to time. And words dont mean the same thing to the same people.

In 1952 Derek Bentley was hanged as an accomplice of murder, during a burglary with his accomplice Christopher Craig, a policeman showed up and Christopher pointed a gun, Derek yelled "Let him have it" meaning "give the gun to the policeman", but Christopher thought it meant to shoot, so he shot and killed the policeman. Derek was considered de-facto the murderer as he gave the order and was sentenced to death.

How do you bootstrap a language if the rules themselves are written in the language itself? It might seem obvious that parts of the English language live outside of it, in the human condition itself: birth, desire, anxiety, love, pain and death. In our society people are exposed to language since they are born, as their brain develops language becomes a primitive intertwined with thought, not all thought is language and not

all language is thought. However the very basic concept distinguishing "you" and "me" seems to be related to language. It allows you to distinguish "you" from "you in the past" and "you in the future", it allows you to build a theory of mind, meaning you can guess what I will do by thinking as if you become me.

Language deprivation is considered "The forbidden experiment", it is one of the most inhumane things that could be done to a human. There are examples of feral children, children that grew up in the wild alone. It seems after a certain age language becomes very hard to learn, and I am not sure the word "learn" is properly used here, in the same way a parrot "learns" a new word, to learn a symbol you need not only to learn its encoding, but also its relations. As you grow you learn how to relate symbols by yourself. You can create new symbols and new relationships alone. However it seems people can do this only if exposed to language before a certain age. And here exposed to language is saying a lot with few words, it means to be exposed to people around you to interact with them and learn from them.

A human, it seems, can be an animal, but an animal can not be a human. Strangely, if you do not expose the human to other humans up to a certain age, it remains an animal.

There are other languages such as mathematics and logic, which are invariant with the human condition, they are the same for all life in the universe, and it is merely a matter of translation from our symbols to their symbols, but the symbolic relationships are the same.

```
* A is A
* nothing is both A and not-A
* everything is either A or not-A
```

There is a famous joke, there are two books "Everything Harvard Business School Teaches You" and "Everything Harvard Business School Does Not Teach You", and people joke, "In those two books I have all human knowledge, past, present and future".

If you look back at "one bit is the ability to distinguish between two universes", those fundamental laws say exactly this, that one bit is indeed one bit.

It is not very easy to build complex logic systems, as you might have loops, and you wont discover those loops until you evaluate the statement itself.

For example: `This sentence is false.`. Or in our select universe:

```
L = select(L, 0, 1)      # L = NOT(L)

t0: L = ?
t1: L = NOT(?) = ?
t2: L = NOT(NOT(?)) ...
```

Reminder what our select does `select(c,a,b)` is `if c then a else b`.

We can construct infinite loop statements on purpose, but if the logic system becomes large enough we might end up defining things in themselves without us knowing. When `time` is introduced things become clearer, lets use `@` to define "previous value", `L = select(@L, 0, 1)` is just shorthand for `L[t] = select(L[t-1], 0, 1)`. Now the statement is quite trivial, but it depends on the initial condition, what was `L` at time=0, which has to be given from outside the system. The logic system is not separate from its evaluator. `this sentence is false` is a great example, you read it and just ignore it, you say its nonsense, and move on, if we however have an evaluator that can not see it is in an infinite loop, it will be stuck until the end of time.

Programming languages are languages that translate human symbols to selects, we could translate symbols to architecture of wires, or to initial signals given to predefined configuration wires, but the idea is the same. One language we use to translate symbols to wires is called `Verilog`, a language used to translate symbols to initial conditions is called `C`. Some languages are made by us so they can best translate human ideas into the machines we can build or have built, others are discovered and live in the space of logic itself, or even in physics itself. There are computers in this world which we can not yet program because we have not yet discovered or created languages to translate human ideas into programs.

There are also languages for non-existent idealized machines, that can never be made, or at least not yet.

Using our very primitive `select` plus a bit of English, we will make an actual `select` computer, that if scaled up can do anything any computer can do.

Our select language is very simple:

```
^x = 0
x = @x ? 0 : 1
```

That is pretty much it.

- PREVIOUS: `@x` means the previous value of `x`.
- INIT: `^x = 0` means @x starts at 0, or the value `@x` at t0 is `0`.
- SELECT: `select(c,a,b)` is `c ? a : b`, you can read it as `if c then a else b`.
- DEFINE: `x =` this defines `x` to be whatever it is on the right, you can think of it 'naming a signal' or 'naming a wire'.
- SEPARATE: `;` ends a definition, exactly as the end of a line does, so short ones can share a line: `a = 0;  b = 1` says the same as writing them one under the other.

Maybe this example will make it clearer.

```
^x = 0

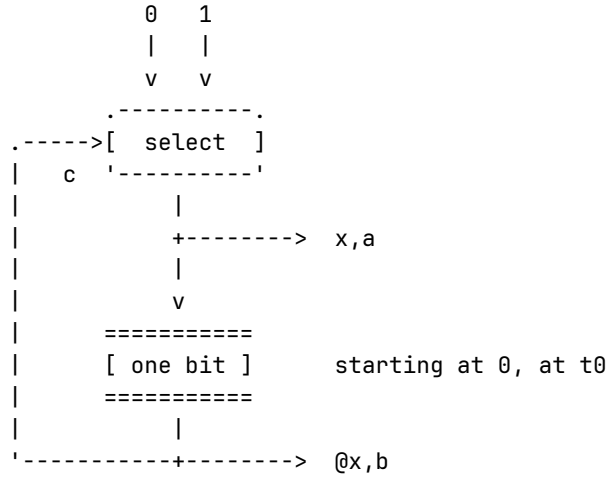
x = @x ? 0 : 1     x toggles every tick
a = x              soldered to x
b = @x             one tick behind x
```

This is how the time evolution looks, `x` and `a` are the same wire.

```
   @x   x   a   b
t1  0   1   1   0
t2  1   0   0   1
t3  0   1   1   0
```

`b = @x` is a bit strange, as if `b` is a wire that is connected to `x` in the past.

This language then gets compiled to a graph of selects.



In our example its only one select, you see = compiles to nothing, its just a way for us to refer to a signal.

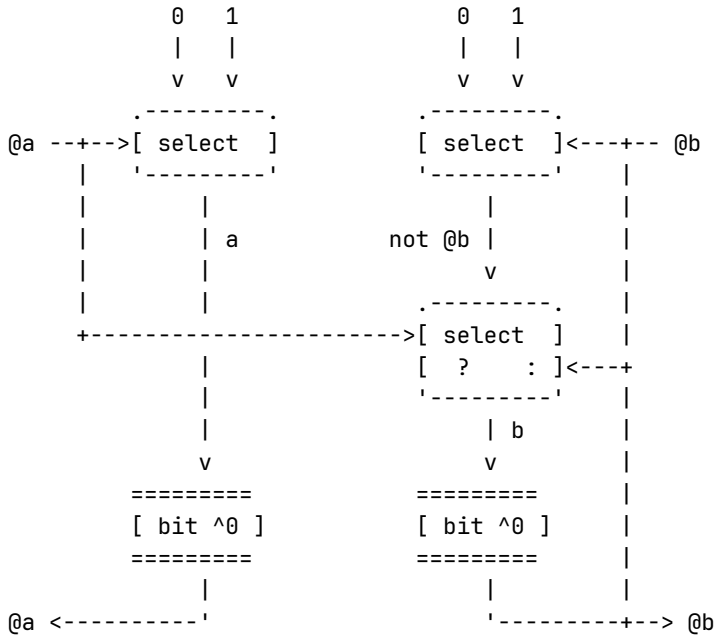
```
^a = 0
^b = 0

a = @a ? 0 : 1                a toggles: xor with 1
b = @a ? (@b ? 0 : 1) : @b    b = xor of the previous a and b

    @b @a    b a
t1   0  0    0 1
t2   0  1    1 0
t3   1  0    1 1
t4   1  1    0 0
t5   0  0    0 1
```

I will leave it to you to find meaning in this program, think, what does it actually do?

And the graph it compiles to:



To compile means to translate the human language into the machine itself, either the wires or the values used by wires. Again, `=` means nothing in our machine, it is just for us, to help us think about how to tell the machine what to do. We can of course just build the graph of selects, but once they become a few hundred signals even the smartest people will struggle to keep track. So, a programming language is a tool for humans to talk to machines. The machines themselves do not need it.

I have used the term program multiple times now, now that you have seen quite a lot of programs, in different forms, do you think `x = @x ? 0 : 1` is the program or the actual wires of the graph are the program, or the initial condition of `x` is the program?

I define a program as the actual machine that evolves through time. Its code: `x = @x ? 0 : 1` is nothing, the compiled graph is nothing, the initial condition `^x = 0` is nothing, but together, they are one machine.

You can compile the same code to a different machine, maybe it uses inverse selects only, maybe its primitive is NAND only, or maybe use a different language that compiles to the same select graph? Those will

be different programs. As you, the programmer, are part of the machine too.

> Every program that is created must have a purpose; if it does not, it is deleted.
> – Rama-Kandra, The Matrix Revolutions.

3 One Minus One

We can do `select`, have memory through time, and add numbers, now what?

Those 3 things are enough to build a computer that can execute any program, from playing netflix to predicting words. In fact, your phone, your laptop, the computers in your car, they can do only those 3 things.

When you are making any kind of machine, you are mainly constrained by your abilities. Your abilities to understand yourself, the world, and the machine. In fact the main question of this book is where does the machine ends and I begin.

I make no distinction between using a machine and programming a machine, and of course the device, its program, and its programmer are together the actual machine.

Lets take a look at of the oldest complex machines we have: `the shaduf`.



Figure 3.1: Shadoof

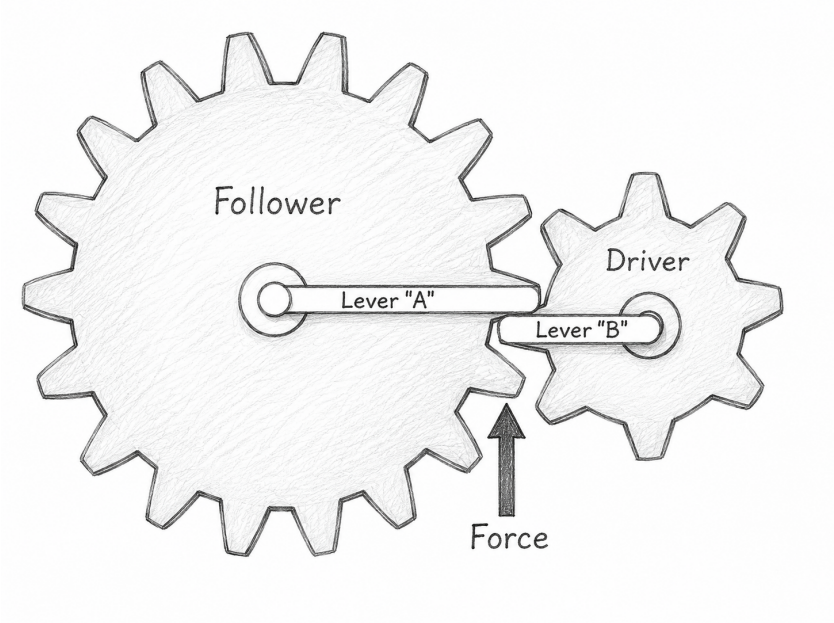
Used in the time of Sargon of Akkad, almost 5000 years ago.



Figure 3.2: King Sargon

How do you think you program this machine? How long is the pole? Where is the pivot? How deep is the well? Even without touching it, you can imagine how it works.

Gears are in fact a circle of levers pushing on each other.



We use series of gears for many things, for example to do torque reduction or multiplication. In your bicycle your pedals rotate a gear, which is pushing on the chain which pushing on another gear, which in turn rotates the tire and you move forward. You know the size of the gears matters.



Figure 3.3: Transmission internals for a 2009 Volkswagen Golf

The shaduf I understand, the gear I understand, but the transmission box I need to sit down on pen and paper and see which wheel turns which wheel and how. It will take me some time to figure it out, so that I can change it. You can say: well you dont need to understand it to use it, but this is incorrect, as you are blind to what it can and can not do. How do you know that you are using it completely and correctly? You can use an airplane for storage, and it is fine storage room, hard to beat, light, mobile, insulated, but you can also use it for flying.

Becuse of my inability to see and reason about a machine as a whole, I break it down in pieces and think about how they interact with each other. In the case of a computer, I think of the memory as one component, as the adder as another, of course we can make memory+adder with `select` and `time`, but it will be difficult for me to reason through each step, and this will limit my ability to build more complicated circuits because I wont be able to pretend I am the machine. This is what it means to understand, is to pretend you are `it`, and think what `it` will do. `it` could be anything, an electron, a bit, a circuit, a man playing chess, or worse, a man playing poker.

We will build our computer with `select` and `time` to be able to execute few primitive operations, then we will build more complicated programs with said operations while we do not think of the underlying circuit of selects.

- read memory
- write memory
- change which memory we execute next
- subtract numbers

3.1 Addressible Memory

Lets make 2 memory cells, with 3 bits each

```
// bit 0 is the LSB, bit 2 the MSB
^cell_0_0 = 0;  ^cell_0_1 = 0;  ^cell_0_2 = 0
^cell_1_0 = 0;  ^cell_1_1 = 0;  ^cell_1_2 = 0

// we need only 1 bit to distinguish
// which cell we want to read/write
^address = 0

// one bit to know if we are writing
// or not
^writing = 0

// data we want to write
^data_0 = 0;  ^data_1 = 0;  ^data_2 = 0

// are we writing to this cell or not
w_0 = writing ? (address ? 0 : 1) : 0
w_1 = writing ? (address ? 1 : 0) : 0

// if we are writing use the data signal
// otherwise the previous cell value
cell_0_0 = w_0 ? data_0 : @cell_0_0
cell_0_1 = w_0 ? data_1 : @cell_0_1
cell_0_2 = w_0 ? data_2 : @cell_0_2
```

```
cell_1_0 = w_1 ? data_0 : @cell_1_0
cell_1_1 = w_1 ? data_1 : @cell_1_1
cell_1_2 = w_1 ? data_2 : @cell_1_2

// read either cell 0 or cell 1
read_0 = address ? @cell_1_0 : @cell_0_0
read_1 = address ? @cell_1_1 : @cell_0_1
read_2 = address ? @cell_1_2 : @cell_0_2
```

This is our whole program, it is just few selects, you can imagine how we can scale this to thousands of bits if we want to, it is just more code, but the idea is the same.

Lets follow it through time:

```
   wrt  addr  d2 d1 d0 |  w0  w1 | cell (2..0)   | read (2..0)
t1   1    1   1  1  0  |  0   1  |   1 1 0       |  0 0 0
t2   0    1   0  0  0  |  0   0  |   1 1 0       |  1 1 0
t3   0    0   0  0  0  |  0   0  |   1 1 0       |  0 0 0
```

On t1 we write `110` to `cell_1`, after that if we set `writing` to `0`, we read can read it back through `read_0-2`, at t3 if we change addr to `0` we read all zeroes because we are reading `cell_0`

I am showing changing `writing` and `addr` just to illustrate what would happen if we could change them, so far our graph of selects does not support that, but we will build on it.

3.2 Subtract

We only know how to add binary numbers, so what is a negative binary number? We encode negative binary numbers in a system called `two's complement`. `-b` is defined as `NOT(b) + 1`. The main reason for this encoding is that we can use the same addercircuit we have.

```
011   3
010   2
001   1
000   0
111  -1
110  -2
101  -3
100  -4

3 - 1 = 3 + -1 = 011 + (NOT(001) + 1)

NOT(001) = 110
 110 + 1 = 111

          1     11    111
  011    011    011    011
+ 111  + 111  + 111  + 111
  ---    ---    ---    ---
           0     10    010

                carry 1 falls off
```

Negating `-4` is `-4` again.

```
NOT(100) = 011
 011 + 1 = 100
```

That is why we go from `3` to `-4`, It is up to your computer how to read a binary pattern `111` could be 7 or `-1`, Or could be a red pixelon your screen, the truth is the eye of the beholder.

In our select computer all numbers will be between `3` and `-4`. So lets build our custom adder. We wont simplify anything, we will build two complete adders, one to add `+1` and one to add `a + -b`.

Reminder how a full adder looks:

```
XOR(a,b)  = select(a, NOT(b), b)
          = select(a, select(b,0, 1), b)
```

```
AND(a, b) = select(a, b, 0)
OR(a, b)  = select(a, 1, b)

FSUM(a,b,c)   = XOR(XOR(a,b), c)
                SUM(SUM(a,b), c)
                a b c   a+b+c
                1 1 1   1
                1 1 0   0
                1 0 1   0
                1 0 0   1
                0 1 1   0
                0 1 0   1
                0 0 1   1
                0 0 0   0

FCARRY(a,b,c) = OR(AND(a,b), AND(c, XOR(a,b)))

s0 = FSUM(a0, b0, 0)     c0 = FCARRY(a0, b0, 0)
s1 = FSUM(a1, b1, c0)    c1 = FCARRY(a1, b1, c0)
s2 = FSUM(a2, b2, c1)    c2 = FCARRY(a2, b2, c1)
```

So, lets do it

```
^a0 = 0;  ^a1 = 0;  ^a2 = 0
^b0 = 0;  ^b1 = 0;  ^b2 = 0

zero = 0;   one_0 = 1;  one_1 = 0;  one_2 = 0

not_b0 = b0 ? 0 : 1;   not_b1 = b1 ? 0 : 1
not_b2 = b2 ? 0 : 1

// ADDER 1:  nb = NOT(b) + 1
// x is XOR(a,b) of this column
// nb is the sum bit, k is the carry going left
x_0  = not_b0 ? (one_0 ? 0 : 1) : one_0
nb0  = x_0 ? (zero ? 0 : 1) : zero
k_0  = (not_b0 ? one_0 : 0) ? 1 : (zero ? x_0 : 0)

x_1  = not_b1 ? (one_1 ? 0 : 1) : one_1
```

```
nb1  = x_1 ? (k_0 ? 0 : 1) : k_0
k_1  = (not_b1 ? one_1 : 0) ? 1 : (k_0 ? x_1 : 0)

x_2  = not_b2 ? (one_2 ? 0 : 1) : one_2
nb2  = x_2 ? (k_1 ? 0 : 1) : k_1
k_2  = (not_b2 ? one_2 : 0) ? 1 : (k_1 ? x_2 : 0)
// k_2 falls off the end, nobody reads it

// ADDER 2:  s = a + nb
y_0  = a0 ? (nb0 ? 0 : 1) : nb0
s0   = y_0 ? (zero ? 0 : 1) : zero
c_0  = (a0 ? nb0 : 0) ? 1 : (zero ? y_0 : 0)

y_1  = a1 ? (nb1 ? 0 : 1) : nb1
s1   = y_1 ? (c_0 ? 0 : 1) : c_0
c_1  = (a1 ? nb1 : 0) ? 1 : (c_0 ? y_1 : 0)

y_2  = a2 ? (nb2 ? 0 : 1) : nb2
s2   = y_2 ? (c_1 ? 0 : 1) : c_1
c_2  = (a2 ? nb2 : 0) ? 1 : (c_1 ? y_2 : 0)
// c_2 falls off the end too
```

Trying it out `3 - 1`, `a = 011`, `b = 001`, so `not_b = 110`

```
col   not_b  one   carry in |  x   nb   carry out
 0      0     1       0     |  1    1       0
 1      1     0       0     |  1    1       0
 2      1     0       0     |  1    1       0

nb = 111   which is -1


col    a    nb   carry in |  y    s   carry out
 0     1     1      0     |  0    0       1
 1     1     1      1     |  0    1       1
 2     0     1      1     |  1    0       1

s = 010   which is 2       c_2 = 1, falls off
```

One thing we have to talk about is which column is which:

```
 0  1  1
b2 b1 b0
```

b0 is the least significant bit, LSB, and b2 is the most significant bit, MSB; Least and most significant means changing this big changes the number most or least, e.g:

```
0   0  0
b2 b1 b0
```

If we change `b0` we get from 0 to 1, if we change `b2` we go to -4, or 4 if we did not have sign.

So, now we have memory and we can add, there is one tiny thing we need in order to build our computer, and that is to make choice based on the memory itself. In this computer we will read 3 numbers at a time, `a`, `b`, and `c`, we will subtract the value at `a` from the value at `b`, store the result in the location of `b`, and if the result is larger than zero we will read the next 3 numbers, if the result is smaller or equal to zero we will move to location `c` and read 3 numbers from there.

Written in pseudo code each instruction does this:

```
mem[b] = mem[b] - mem[a]
if (mem[b] ⩽ 0)
    goto c
```

This is a single instruction computer, called SUBLEQ (subtract and branch if less than or equal to zero).

To check if the result is smaller or equal than zero is quite easy for us:

```
leq = s2 ? 1 : (s0 ? 0 : (s1 ? 0 : 1))
```

If the most significant bit is one, then we are good, means the number is negative, otherwise all the other bits must be zero.

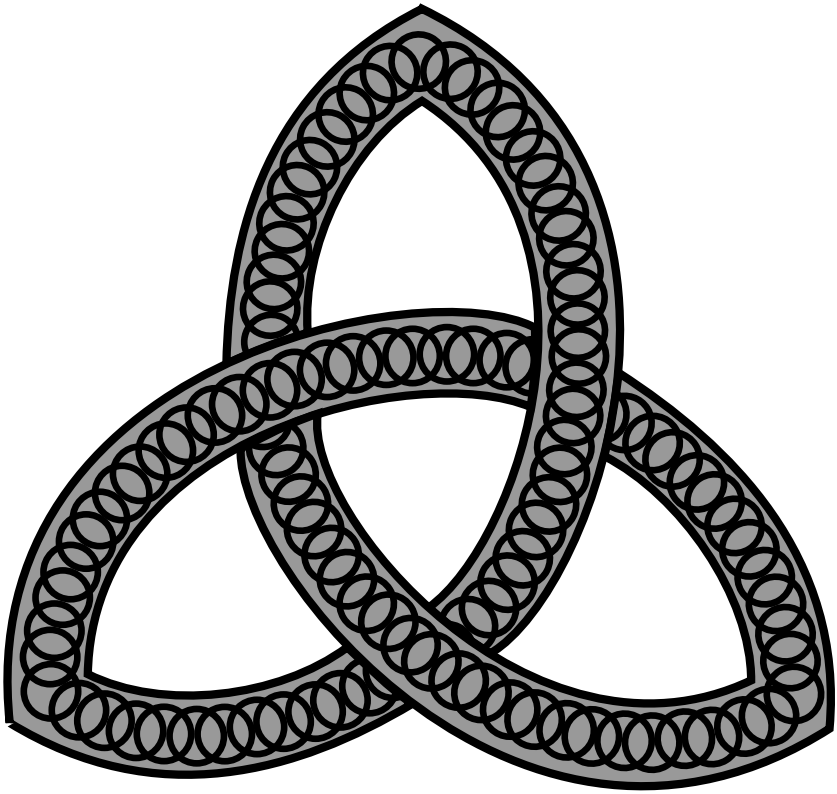
3.3 Clock

We want to know what we are doing each step, for that we will create a phase clock.

```
^p0 = 0;
^p1 = 0;
^p2 = 0;
^p3 = 0;
^p4 = 1

p0 = @p4;
p1 = @p0;
p2 = @p1;
p3 = @p2;
p4 = @p3
```

Every tick we cycle the bit through `p0 p1 p2 p3 p4`. It is really interesting to think how the "wires" of this phase ring look, as every signal is connected to the past of the previous signal. I imagine it like a celtic knot.



3.4 Computer

Ok, lets build the whole computer, and then we will use it.

```
// ---------------- memory: 8 cells, 3 bits, holding the program

^cell_0_0 = 0; ^cell_0_1 = 1; ^cell_0_2 = 1  // 6
^cell_1_0 = 1; ^cell_1_1 = 1; ^cell_1_2 = 1  // 7
^cell_2_0 = 1; ^cell_2_1 = 1; ^cell_2_2 = 0  // 3
^cell_3_0 = 0; ^cell_3_1 = 1; ^cell_3_2 = 1  // 6
```

```
^cell_4_0 = 0; ^cell_4_1 = 1; ^cell_4_2 = 1  // 6
^cell_5_0 = 1; ^cell_5_1 = 1; ^cell_5_2 = 0  // 3
^cell_6_0 = 1; ^cell_6_1 = 0; ^cell_6_2 = 0  // 1
^cell_7_0 = 1; ^cell_7_1 = 1; ^cell_7_2 = 0  // 3

// ---------------- the phase ring, exactly one bit hot
// the hat marks the phase before tick 1, so p0 acts first

^p0 = 0;  ^p1 = 0;  ^p2 = 0;  ^p3 = 0;  ^p4 = 1

p0 = @p4;  p1 = @p0;  p2 = @p1;  p3 = @p2;  p4 = @p3

// ---------------- registers

^pc0 = 0;   ^pc1 = 0;   ^pc2 = 0

^t0 = 0;    ^t1 = 0;    ^t2 = 0
^rb0 = 0;   ^rb1 = 0;   ^rb2 = 0
^rc0 = 0;   ^rc1 = 0;   ^rc2 = 0

// p0 p1 p2 read the program, p3 p4 read the data
fetching = p0 ? 1 : (p1 ? 1 : p2)

// ---------------- where we point

addr_0 = fetching ? @pc0 : (p3 ? @t0 : @rb0)
addr_1 = fetching ? @pc1 : (p3 ? @t1 : @rb1)
addr_2 = fetching ? @pc2 : (p3 ? @t2 : @rb2)

// ---------------- read

read_0 = (addr_2 ? (addr_1 ? (addr_0 ? @cell_7_0 : @cell_6_0)
                           : (addr_0 ? @cell_5_0 : @cell_4_0))
                 : (addr_1 ? (addr_0 ? @cell_3_0 : @cell_2_0)
                           : (addr_0 ? @cell_1_0 : @cell_0_0)))

read_1 = (addr_2 ? (addr_1 ? (addr_0 ? @cell_7_1 : @cell_6_1)
                           : (addr_0 ? @cell_5_1 : @cell_4_1))
                 : (addr_1 ? (addr_0 ? @cell_3_1 : @cell_2_1)
                           : (addr_0 ? @cell_1_1 : @cell_0_1)))
```

```
read_2 = (addr_2 ? (addr_1 ? (addr_0 ? @cell_7_2 : @cell_6_2)
                           : (addr_0 ? @cell_5_2 : @cell_4_2))
                 : (addr_1 ? (addr_0 ? @cell_3_2 : @cell_2_2)
                           : (addr_0 ? @cell_1_2 : @cell_0_2)))

// ---------------- latch the operands

// t takes the operand a at p0, and at p3 the value found at that
// address. the name and the thing it names are never both needed,
// so one register holds each in turn and there is no a register
load_t = p0 ? 1 : p3

t0 = load_t ? read_0 : @t0
t1 = load_t ? read_1 : @t1
t2 = load_t ? read_2 : @t2

rb0 = p1 ? read_0 : @rb0
rb1 = p1 ? read_1 : @rb1
rb2 = p1 ? read_2 : @rb2

rc0 = p2 ? read_0 : @rc0
rc1 = p2 ? read_1 : @rc1
rc2 = p2 ? read_2 : @rc2

// ---------------- s = read - t, the two adders from above
// t is the adder's subtrahend, so not_t is NOT(t); it has nothing
// to do with the operand register rb

zero = 0;   one_0 = 1;  one_1 = 0;  one_2 = 0

not_t0 = @t0 ? 0 : 1;   not_t1 = @t1 ? 0 : 1
not_t2 = @t2 ? 0 : 1

// ADDER 1:  nb = NOT(t) + 1
x_0  = not_t0 ? (one_0 ? 0 : 1) : one_0
nb0  = x_0 ? (zero ? 0 : 1) : zero
k_0  = (not_t0 ? one_0 : 0) ? 1 : (zero ? x_0 : 0)

x_1  = not_t1 ? (one_1 ? 0 : 1) : one_1
```

```
nb1  = x_1 ? (k_0 ? 0 : 1) : k_0
k_1  = (not_t1 ? one_1 : 0) ? 1 : (k_0 ? x_1 : 0)

x_2  = not_t2 ? (one_2 ? 0 : 1) : one_2
nb2  = x_2 ? (k_1 ? 0 : 1) : k_1
// there is no k_2: the carry out of the top bit goes nowhere

// ADDER 2:  s = read + nb
y_0  = read_0 ? (nb0 ? 0 : 1) : nb0
s0   = y_0 ? (zero ? 0 : 1) : zero
c_0  = (read_0 ? nb0 : 0) ? 1 : (zero ? y_0 : 0)

y_1  = read_1 ? (nb1 ? 0 : 1) : nb1
s1   = y_1 ? (c_0 ? 0 : 1) : c_0
c_1  = (read_1 ? nb1 : 0) ? 1 : (c_0 ? y_1 : 0)

y_2  = read_2 ? (nb2 ? 0 : 1) : nb2
s2   = y_2 ? (c_1 ? 0 : 1) : c_1
// and no c_2 either, for the same reason

leq = s2 ? 1 : (s0 ? 0 : (s1 ? 0 : 1))

// ---------------- the choice

branch = p4 ? leq : 0

// adding 1: the bottom bit flips, and the carry into each higher
// bit is every bit below it, and-ed together
inc0 = @pc0 ? 0 : 1
inc1 = @pc1 ? (@pc0 ? 0 : 1) : @pc0
kc1  = @pc1 ? @pc0 : 0
inc2 = @pc2 ? (kc1 ? 0 : 1) : kc1

pc0 = fetching ? inc0 : (branch ? @rc0 : @pc0)
pc1 = fetching ? inc1 : (branch ? @rc1 : @pc1)
pc2 = fetching ? inc2 : (branch ? @rc2 : @pc2)

// ---------------- write

writing = p4
```

```
w_0 = writing ? (addr_2 ? 0 : (addr_1 ? 0 : (addr_0 ? 0 : 1))) : 0
w_1 = writing ? (addr_2 ? 0 : (addr_1 ? 0 : (addr_0 ? 1 : 0))) : 0
w_2 = writing ? (addr_2 ? 0 : (addr_1 ? (addr_0 ? 0 : 1) : 0)) : 0
w_3 = writing ? (addr_2 ? 0 : (addr_1 ? (addr_0 ? 1 : 0) : 0)) : 0
w_4 = writing ? (addr_2 ? (addr_1 ? 0 : (addr_0 ? 0 : 1)) : 0) : 0
w_5 = writing ? (addr_2 ? (addr_1 ? 0 : (addr_0 ? 1 : 0)) : 0) : 0
w_6 = writing ? (addr_2 ? (addr_1 ? (addr_0 ? 0 : 1) : 0) : 0) : 0
w_7 = writing ? (addr_2 ? (addr_1 ? (addr_0 ? 1 : 0) : 0) : 0) : 0

cell_0_0 = w_0 ? s0 : @cell_0_0
cell_0_1 = w_0 ? s1 : @cell_0_1
cell_0_2 = w_0 ? s2 : @cell_0_2

cell_1_0 = w_1 ? s0 : @cell_1_0
cell_1_1 = w_1 ? s1 : @cell_1_1
cell_1_2 = w_1 ? s2 : @cell_1_2

cell_2_0 = w_2 ? s0 : @cell_2_0
cell_2_1 = w_2 ? s1 : @cell_2_1
cell_2_2 = w_2 ? s2 : @cell_2_2

cell_3_0 = w_3 ? s0 : @cell_3_0
cell_3_1 = w_3 ? s1 : @cell_3_1
cell_3_2 = w_3 ? s2 : @cell_3_2

cell_4_0 = w_4 ? s0 : @cell_4_0
cell_4_1 = w_4 ? s1 : @cell_4_1
cell_4_2 = w_4 ? s2 : @cell_4_2

cell_5_0 = w_5 ? s0 : @cell_5_0
cell_5_1 = w_5 ? s1 : @cell_5_1
cell_5_2 = w_5 ? s2 : @cell_5_2

cell_6_0 = w_6 ? s0 : @cell_6_0
cell_6_1 = w_6 ? s1 : @cell_6_1
cell_6_2 = w_6 ? s2 : @cell_6_2

cell_7_0 = w_7 ? s0 : @cell_7_0
cell_7_1 = w_7 ? s1 : @cell_7_1
```

```
cell_7_2 = w_7 ? s2 : @cell_7_2
```

Lets follow it through. The program loaded at ^ is:

```
0 1 2 3 4 5 6 7
---------------
6 7 3 6 6 3 1 3

mem[b] = mem[b] - mem[a]
if (mem[b] ⩽ 0)
    goto c
```

The first instruction is `6 7 3`:

```
mem[7] = mem[7] - mem[6]
              3 - 1
         mem[7] = 2
if (2 ⩽ 0)
    goto 3
```

2 is not less or equal than zero, so we continue, it so happens the next instruction is on address `3 4 5` anyway.

```
0 1 2 3 4 5 6 7
---------------
6 7 3 6 6 3 1 2
```

The next instruction is `6 6 3`, in this instruction both `a` and `b` are the same location.

```
mem[6] = mem[6] - mem[6]
              1 - 1
         mem[6] = 0
if (0 ⩽ 0)
    goto 3
```

After that our memory looks like this:

```
0 1 2 3 4 5 6 7
---------------
6 7 3 6 6 3 0 2
```

And we will spin at `6 6 3` forever. This program is quite silly, we are using it just to illustrate how the computer works, in this time diagram we group signals together, so `cell_0_[0 1 2]` are just shown as `cell_0`.

Those are the first 12 steps:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
p	1	2	4	8	16	1	2	4	8	16	1	2
fetching	1	1	1	0	0	1	1	1	0	0	1	1
addr	0	1	2	6	7	3	4	5	6	6	3	4
read	6	7	3	1	3	6	6	3	1	1	6	6
load_t	1	0	0	1	0	1	0	0	1	0	1	0
t	6	6	6	1	1	6	6	6	1	1	6	6
rb	0	7	7	7	7	7	6	6	6	6	6	6
rc	0	0	3	3	3	3	3	3	3	3	3	3
zero	.	.	.	.	0	.	.	.	.	0	.	.
one	.	.	.	.	1	.	.	.	.	1	.	.
not_t	.	.	.	.	6	.	.	.	.	6	.	.
x	.	.	.	.	7	.	.	.	.	7	.	.
nb	.	.	.	.	7	.	.	.	.	7	.	.
k	.	.	.	.	0	.	.	.	.	0	.	.
y	.	.	.	.	4	.	.	.	.	6	.	.
s	.	.	.	.	2	.	.	.	.	0	.	.
c	.	.	.	.	3	.	.	.	.	3	.	.
leq	.	.	.	.	0	.	.	.	.	1	.	.
branch	.	.	.	0	0	.	.	.	0	1	.	.
inc	1	2	3	.	.	4	5	6	.	.	4	5
kc1	0	0	0	.	.	1	0	0	.	.	1	0
pc	1	2	3	3	3	4	5	6	6	3	4	5
writing	0	0	0	0	1	0	0	0	0	1	0	0
w	0	0	0	0	128	0	0	0	0	64	0	0
cell_0	6	6	6	6	6	6	6	6	6	6	6	6
cell_1	7	7	7	7	7	7	7	7	7	7	7	7
cell_2	3	3	3	3	3	3	3	3	3	3	3	3
cell_3	6	6	6	6	6	6	6	6	6	6	6	6
cell_4	6	6	6	6	6	6	6	6	6	6	6	6

```
  cell_5   3   3   3   3   3   3   3   3   3   3   3   3
  cell_6   1   1   1   1   1   1   1   1   1   0   0   0
  cell_7   3   3   3   3   2   2   2   2   2   2   2   2

. never asked
```

It might not look like it, but this computer can execute any program that any other computer can execute. For educational purposes our computer is very tiny, and it is borderline useless, but imagine the cells are 64 bit wide and we have billions or even trillions of them. Even if we can do only one instruction, our computer can do what any other computer can do, most things will be slower because we can do only one kind of instruction, but the result of the computation will be the same.

So far we used `select` to build a computer that can do arithmetic, as if `select` is more primitive than arithmetic, but if you have +, -, *, you can do select with arithmetic.

```
select(c,a,b) = ca+(1-c)b
```

Just a snippet of how our computer would look without `select`

```
leq = s2*1 + (1-s2)*(s0*0 + (1-s0)*(s1*0 + (1-s1)*1))

branch = p4*leq + (1-p4)*0

inc0 = @pc0*0 + (1-@pc0)*1
inc1 = @pc1*(@pc0*0 + (1-@pc0)*1) + (1-@pc1)*@pc0
kc1 = @pc1*@pc0 + (1-@pc1)*0
inc2 = @pc2*(kc1*0 + (1-kc1)*1) + (1-@pc2)*kc1

pc0 = fetching*inc0 + (1-fetching)*(branch*@rc0 + (1-branch)*@pc0)
pc1 = fetching*inc1 + (1-fetching)*(branch*@rc1 + (1-branch)*@pc1)
pc2 = fetching*inc2 + (1-fetching)*(branch*@rc2 + (1-branch)*@pc2)
```

In fact, `NAND(a,b)` is the same as `1 - a*b`, and `select(c,a,b)`:

```
select(c,a,b) = NAND(
                      NAND(c,a),
                      NAND(NAND(c,c),b)
                  )
```

`NAND(a,b)` is also `NOT(a*b)`, `AND(a,b)` is `a*b`.

```
NOT(a)   = NAND(a,a)
AND(a,b) = NOT(
               NAND(a,b)
            )

OR(a,b)  = NAND(
                NAND(a,a),
                NAND(b,b)
            )

XOR(a,b) = NAND(
                NAND(a, NAND(a,b)),
                NAND(b, NAND(a,b))
            )
```

Or the other way around:

```
NAND(a,b) = NOT(
                AND(a,b)
             )
```

Or again with selects

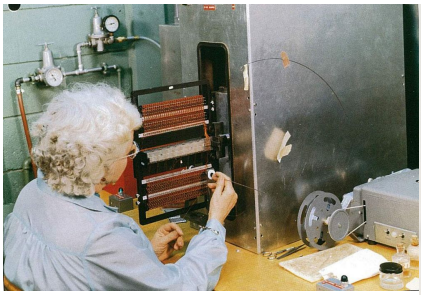
```
NOT(a)    = select(a, 0, 1)
AND(a, b) = select(a, b, 0)
OR(a, b)  = select(a, 1, b)
XOR(a,b)  = select(a, NOT(b), b) = select(a, select(b,0, 1), b)
NAND(a,b) = NOT(AND(a,b)) = select(select(a,b,0), 0, 1)
```

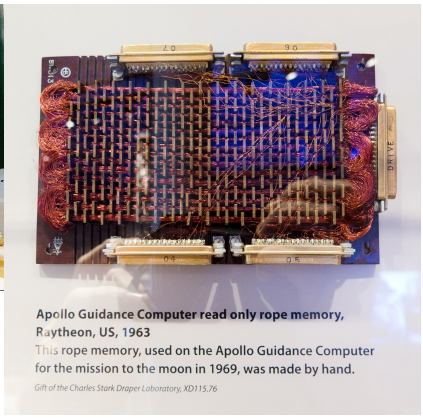
All those things are the same thing.

Remember this, because we will come back to it once we do `select` with neural networks, and neural networks with `select`.

3.5 Assembly Language

Lets actually build something useful with our `select` machine, we want to create proper programs, that take input and output and do useful things, like speak or listen. When the machine is tiny we can imagine we are it, in fact in the 1950s and 1960s people used to literally "weave" memory with wire and needles.





Apollo Guidance Computer read only rope memory, Raytheon, US, 1963
This rope memory, used on the Apollo Guidance Computer for the mission to the moon in 1969, was made by hand.
Gift of the Charles Stark Draper Laboratory, XD115.76

People actually went to the moon with programs weaved by human hands.

But, we want to build programs with billions of SUBLEQ instructions, it is impossible (for me) to think of it in terms of `select` or even in terms of `SUBLEQ` and memory locations. I want to be able to refer to memory address by a label, without me knowing where it is, because I actually do not care, I care how I relate this memory to the rest of the program, but not where it actually is.

For example, counting down from 5, I do not care where the number 5 lives, I just want to decrement it with 1, which I also do not care where it lives, nor where the actual instructions live.

In raw SUBLEQ machine code the counter looks like this:

```
5 5 6  5 1 0  4 3 12  5 5 6  5 5 12
```

In my head I think about it very differently:

```
SUBLEQ zero, zero, keep_counting     // goto keep_counting

counter: 5                           // counter=5
one: 1                               // one=1
zero: 0                              // zero=0

keep_counting:
  SUBLEQ one, counter, done          // counter = counter - 1
                                     // if counter ⩽0 goto done
  SUBLEQ zero, zero, keep_counting   // goto keep_counting
done:
  SUBLEQ zero, zero, ?               // halt
```

When the program starts I want to jump to my counting code, I want to have the counter set to 5, and number one somewhere in memory, I also want the number zero somewhere to use it to do jumps. `?` means wherever this instruction is, because I want when I am done to jump to itself, I could also write `done: SUBLEQ zero, zero, done` but it is very useful to refer to the current instruction's address.

```
0000       5      5      6    zero zero keep_counting
0003       5                  counter: 5
0004       1                  one: 1
0005       0                  zero: 0
0006       4      3     12    one counter done
0009       5      5      6    zero zero keep_counting
0012       5      5     12    zero zero ?
```

This language is called an assembly language, it is basically machine code with labels. It gets compiled to SUBLEQ, which gets compiled to selects which gets compiled to physics. It allows you to abstract away the addressess themselves.

Our assembly has the following syntax, everything is read line by line, split by whitespace.

```
1) 5                -  a data cell
2) count Z          - an instruction, C is the next line, ?+3
3) one counter done - an instruction, A B C
```

```
4) 1 2 3 4          - error

? addresses the first cell of whatever the line emits,
?+1 ?+2 ?-1 ?-2 and so on can address ahead or behind

abc: defines a label "abc" binding to the next address
we can refer to that address in the program.



macro add a b
  a Z
  Z b
  Z Z
end

macro defines a chunk of code that is substituted
whenever the macro is used.

.abc: defines label that gets unique id
every time the macro is expanded (.abc__1 .abc__2...)

macro jmp c
  Z Z c
end

jmp done // comment
everything after // is ignored
```

Because C is the next line, a two operand instruction lands in the same place whether the test passes or fails. It is unconditional.

Our macro library begins with three lines of its own, and every program is assembled after them. The machine starts at address 0, so address 0 has to be an instruction, and this jump is it; `Z` and `t` are just added after.

```
0000       3      3      7   Z Z start  // jmp start
0003       0                 Z: 0
0004       0                 t: 0
...
```

`Z` is just a normal cell where we have to make sure it stays zero by convention, it is in no way different than other memory, `t` is just a scratch temporary cell we can use however we want.

We can build quite handy macros out of the one instruction: sub, zero, jmp, jle, halt, add, mov, neg, load and store.

```
macro sub a b
  a b
end

macro zero b
  b b
end

macro jmp c
  Z Z c
end

macro jle b c
    Z b c
end

macro halt
    Z Z ?
end

macro add a b
  a Z
  Z b
  Z Z
end

macro mov a b
  zero b
  add a b
end

macro neg b
  zero t
  sub b t
  zero b
  add t b
end

// pointers, both self modifying code

// mem[dst] = mem[mem[p]]
macro load p dst
  mov p .site      // patch the A slot: .site cell now holds mem[p]
  zero t           // t = 0
  .site: 0 t       // executes as: t = t - mem[mem[p]]
  zero dst         // dst = 0
  sub t dst        // dst = dst - t = mem[mem[p]]
end

// mem[mem[p]] = mem[src]
macro store p src
   mov p .zap       // patch three operand slots with mem[p]
   mov p .zap1
   mov p .put1
   zero t
   sub src t        // t = -mem[src]
```

```
   .zap: 0          // \
   .zap1: 0         //  } executes as: mem[p] mem[p] ?+1
   ?+1              // /   mem[mem[p]] -= mem[mem[p]] target zeroed
   .put: t          // \
   .put1: 0         //  } executes as: t mem[p] ?+1
   ?+1              // /   mem[mem[p]] -= mem[t] =
 end                      0 - (-mem[src]) = mem[src]

'ab' is compiled to 97 98 strings are compiled to their ASCII values
character by character
```

I havent spoken about the ASCII encoding yet, it is just an agreed way between humans how to encode latin characters and symbols. A is 65, a is 97.

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u

```
22   SYN      54     6       86    V      118    v
23   ETB      55     7       87    W      119    w
24   CAN      56     8       88    X      120    x
25   EM       57     9       89    Y      121    y
26   SUB      58     :       90    Z      122    z
27   ESC      59     ;       91    [      123    {
28   FS       60     <       92    \      124    |
29   GS       61     =       93    ]      125    }
30   RS       62     >       94    ^      126    ~
31   US       63     ?       95    _      127   DEL
```

Now our counter program becomes:

```
count:  5
one:    1

start:
  jle count done
  sub one count
  jmp start

done:
  halt
```

And this is what the assembler makes of it:

```
0000       3      3      7   Z Z start     // jmp start
0003       0                 Z: 0
0004       0                 t: 0
0005       5                 count:  5
0006       1                 one:    1
0007       3      5     16   Z count done  // jle count done
0010       6      5     13   one count     // sub one count
0013       3      3      7   Z Z start     // jmp start
0016       3      3     16   Z Z ?         // halt
```

In the end the program is `3 3 7 0 0 5 1 3 5 16 6 5 13 3 3 7 3 3 16`, we can weave it in core memory, neural network weights, or grind it

in billiard balls computer's table boundary, if the machine can execute subleq instructions they will do the same thing. ChatGPT has no issue thinking about the program in terms of `3 3 7..`, but I do, unless I have pen and paper, some milk and cookies, and an afternoon of patience.

Using this assembly language we will build even better language for humans, it is called Forth, possibly the smallest and most powerful language I know. And possibly the only computer language that I can call "language" in the true sense of the word.

3.6 Input and Output

Our `select(c,a,b)` SUBLEQ machine does not have input or output support, everything is predetermined by the initial conditions, to have input and output means to connect the machine to the machine of the universe. Outside observer can look at any signal that we have and get its value, it is free for us, we just have to agree with them where to look so we know to store the values there. Input however requires a change in our select language. `in = ? ?` means in this time moment if the value is realized it comes from outside. We do not know what gives us the input or ouptut, is it another machine, is it determined or random or oracular. We just know it is "outside" of our machine.

To be realized means to be evaluated this instant, to be needed so the evaluator knows how to continue, e.g. given `in = ?`, `select(in, a, b)` we must ask, `select(1, a, in)` we dont.

In our subleq computer needs a small change to support I/O. Addresss -1, all ones 111 becomes our io port.

```
in_0 = ?
in_1 = ?
in_2 = ?

read_0 = (
  addr_2
    ? addr_1
        ? (addr_0 ? (p3 ? in_0 : 0) : @cell_6_0)
        : (addr_0 ? @cell_5_0 : @cell_4_0)
```

```
    : addr_1
        ? (addr_0 ? @cell_3_0 : @cell_2_0)
        : (addr_0 ? @cell_1_0 : @cell_0_0)
)
...
```

And in our "idealized" 64 bit subleaq, we take the `-1` cell again.

Notice we did not change `w_7`, we just deleted `cell_7` so we expect the observer to read the `w_7` signal.

You are right to squint a bit, if the machine has a wire to the outside, how can we separate it? What if the machine that connects to our machine is also connected to another machine and it to another and so on, arent all those one machine?

For now, lets draw the boundary at the io port, "the outside", as oposed to our graph of selects plus their initial values at t0 "the inside".

This is the whole code for the SUBLEQ computer with I/O port:

```
^cell_0_0 = 1; ^cell_0_1 = 1; ^cell_0_2 = 1  // 7
^cell_1_0 = 0; ^cell_1_1 = 0; ^cell_1_2 = 1  // 4
^cell_2_0 = 0; ^cell_2_1 = 0; ^cell_2_2 = 0  // 0
^cell_3_0 = 0; ^cell_3_1 = 0; ^cell_3_2 = 0  // 0
^cell_4_0 = 1; ^cell_4_1 = 0; ^cell_4_2 = 0  // 1
^cell_5_0 = 0; ^cell_5_1 = 0; ^cell_5_2 = 0  // 0
^cell_6_0 = 0; ^cell_6_1 = 0; ^cell_6_2 = 0  // 0


^p1 = 0;  ^p2 = 0;  ^p3 = 0;  ^p4 = 0;  ^p5 = 1
p1 = @p5;  p2 = @p1;  p3 = @p2;  p4 = @p3;  p5 = @p4

^ip_0 = 0;  ^ip_1 = 0;  ^ip_2 = 0

^t_0 = 0;   ^t_1 = 0;   ^t_2 = 0
^b_0 = 0;   ^b_1 = 0;   ^b_2 = 0
^c_0 = 0;   ^c_1 = 0;   ^c_2 = 0

// p1 p2 p3 read the program, p4 p5 read the data
walk = p1 ? 1 : (p2 ? 1 : p3)
```

```
plus1_0       = @ip_0 ? 0 : 1
plus1_1       = @ip_1 ? (@ip_0 ? 0 : 1) : @ip_0
plus1_carry_2 = @ip_1 ? @ip_0 : 0
plus1_2       = @ip_2 ? (plus1_carry_2 ? 0 : 1) : plus1_carry_2


address_0 = walk ? @ip_0 : (p4 ? @t_0 : @b_0)
address_1 = walk ? @ip_1 : (p4 ? @t_1 : @b_1)
address_2 = walk ? @ip_2 : (p4 ? @t_2 : @b_2)

read_0 = (
  address_2
    ? address_1
        ? (address_0 ? (p4 ? in_0 : 0) : @cell_6_0)
        : (address_0 ? @cell_5_0 : @cell_4_0)
    : address_1
        ? (address_0 ? @cell_3_0 : @cell_2_0)
        : (address_0 ? @cell_1_0 : @cell_0_0)
)
read_1 = (
  address_2
    ? address_1
        ? (address_0 ? (p4 ? in_1 : 0) : @cell_6_1)
        : (address_0 ? @cell_5_1 : @cell_4_1)
    : address_1
        ? (address_0 ? @cell_3_1 : @cell_2_1)
        : (address_0 ? @cell_1_1 : @cell_0_1)
)
read_2 = (
  address_2
    ? address_1
        ? (address_0 ? (p4 ? in_2 : 0) : @cell_6_2)
        : (address_0 ? @cell_5_2 : @cell_4_2)
    : address_1
        ? (address_0 ? @cell_3_2 : @cell_2_2)
        : (address_0 ? @cell_1_2 : @cell_0_2)
)

load_t = p1 ? 1 : p4
```

```
t_0    = load_t ? read_0 : @t_0
t_1    = load_t ? read_1 : @t_1
t_2    = load_t ? read_2 : @t_2
b_0    = p2 ? read_0 : @b_0
b_1    = p2 ? read_1 : @b_1
b_2    = p2 ? read_2 : @b_2
c_0    = p3 ? read_0 : @c_0
c_1    = p3 ? read_1 : @c_1
c_2    = p3 ? read_2 : @c_2

nt_0   = @t_0 ? 0 : 1
nt_1   = @t_1 ? 0 : 1
nt_2   = @t_2 ? 0 : 1
x_0    = read_0 ? (nt_0 ? 0 : 1) : nt_0
diff_0 = x_0 ? 0 : 1
k_1    = read_0 ? 1 : nt_0
x_1    = read_1 ? (nt_1 ? 0 : 1) : nt_1
diff_1 = x_1 ? (k_1 ? 0 : 1) : k_1
k_2    = x_1 ? k_1 : read_1
x_2    = read_2 ? (nt_2 ? 0 : 1) : nt_2
diff_2 = x_2 ? (k_2 ? 0 : 1) : k_2

nonzero = diff_0 ? 1 : (diff_1 ? 1 : diff_2)
leq     = diff_2 ? 1 : (nonzero ? 0 : 1)

taken = p5 ? leq : 0
ip_0  = walk ? plus1_0 : (taken ? @c_0 : @ip_0)
ip_1  = walk ? plus1_1 : (taken ? @c_1 : @ip_1)
ip_2  = walk ? plus1_2 : (taken ? @c_2 : @ip_2)

at_lo_0   = @b_0 ? 0 : 1
at_lo_1   = @b_0
at_hi_0   = (@b_1 ? 0 : 1) ? (@b_2 ? 0 : 1) : 0
at_hi_1   = @b_1 ? (@b_2 ? 0 : 1) : 0
at_hi_2   = (@b_1 ? 0 : 1) ? @b_2 : 0
at_hi_3   = @b_1 ? @b_2 : 0
wlo_0     = p5 ? at_lo_0 : 0
wlo_1     = p5 ? at_lo_1 : 0
writing_0 = wlo_0 ? at_hi_0 : 0
writing_1 = wlo_1 ? at_hi_0 : 0
```

```
writing_2 = wlo_0 ? at_hi_1 : 0
writing_3 = wlo_1 ? at_hi_1 : 0
writing_4 = wlo_0 ? at_hi_2 : 0
writing_5 = wlo_1 ? at_hi_2 : 0
writing_6 = wlo_0 ? at_hi_3 : 0
writing_7 = wlo_1 ? at_hi_3 : 0

cell_0_0 = writing_0 ? diff_0 : @cell_0_0
cell_0_1 = writing_0 ? diff_1 : @cell_0_1
cell_0_2 = writing_0 ? diff_2 : @cell_0_2
cell_1_0 = writing_1 ? diff_0 : @cell_1_0
cell_1_1 = writing_1 ? diff_1 : @cell_1_1
cell_1_2 = writing_1 ? diff_2 : @cell_1_2
cell_2_0 = writing_2 ? diff_0 : @cell_2_0
cell_2_1 = writing_2 ? diff_1 : @cell_2_1
cell_2_2 = writing_2 ? diff_2 : @cell_2_2
cell_3_0 = writing_3 ? diff_0 : @cell_3_0
cell_3_1 = writing_3 ? diff_1 : @cell_3_1
cell_3_2 = writing_3 ? diff_2 : @cell_3_2
cell_4_0 = writing_4 ? diff_0 : @cell_4_0
cell_4_1 = writing_4 ? diff_1 : @cell_4_1
cell_4_2 = writing_4 ? diff_2 : @cell_4_2
cell_5_0 = writing_5 ? diff_0 : @cell_5_0
cell_5_1 = writing_5 ? diff_1 : @cell_5_1
cell_5_2 = writing_5 ? diff_2 : @cell_5_2
cell_6_0 = writing_6 ? diff_0 : @cell_6_0
cell_6_1 = writing_6 ? diff_1 : @cell_6_1
cell_6_2 = writing_6 ? diff_2 : @cell_6_2

in_0 = ?
in_1 = ?
in_2 = ?
```

We add one more phase to read or write to the io port.

remember our instruction:

```
A B C
res = mem[B] - mem[A]
mem[B] = res
```

```
if res ⩽ 0:
  goto C
```

Reading input is `-1 B C`, this is how the program `-1 5 0` look, which means:

```
res = mem[5] - mem[-1]
mem[5] = res
if res ⩽ 0:
  goto 0
```

The input will be `1 2 3`, this is how the execution looks:

```
         t1  t2  t3  t4  t5  t6  t7  t8  t9 t10 t11 t12 t13 t14 t15
      p   2   4   8  16  32   2   4   8  16  32   2   4   8  16  32
address   0   1   2   7   5   0   1   2   7   5   0   1   2   7   5
   read   7   5   0   1   0   7   5   0   2   7   7   5   0   3   5
      t   7   7   7   1   1   7   7   7   2   2   7   7   7   3   3
      b   0   5   5   5   5   5   5   5   5   5   5   5   5   5   5
      c   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   diff   7   6   1   2   7   6   6   1   3   5   5   6   1   4   2
    leq   1   1   0   0   1   1   1   0   0   1   1   1   0   1   0
  taken   0   0   0   0   1   0   0   0   0   1   0   0   0   0   0
     ip   1   2   3   3   0   1   2   3   3   0   1   2   3   3   3
writing   0   0   0   0  32   0   0   0   0  32   0   0   0   0  32
 cell_5   0   0   0   0   7   7   7   7   7   5   5   5   5   5   2
     in   .   .   .   1   .   .   .   .   2   .   .   .   .   3   .
```

We print unsigned decimals, meaning 101 is 5, even though in our subleq computer this is -3.

You notice we input 1 but we write 7, for 2 we write 5 and for 3 we write 2, its because the instruction executes `mem[5] - input`, at start `mem[5]` is 0, and `0 - 1` in two's complement is `111`. then again we do `mem[5] - input`, but now mem[5] is -1 and the input is 2, so we get -3, and then the next tick we get -6 which does not fit in 2 bits so it raps around to +2.

Now lets do `5 -1 0 0 0 1`

```
res = mem[-1] - mem[5]
mem[-1] = res
if res ⩽ 0:
  goto 0
```

We take what we have in address 5 and put it on the writing signal.

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15
p	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
address	0	1	2	5	7	0	1	2	5	7	0	1	2	5	7
read	5	7	0	1	0	5	7	0	1	0	5	7	0	1	0
t	5	5	5	1	1	5	5	5	1	1	5	5	5	1	1
b	0	7	7	7	7	7	7	7	7	7	7	7	7	7	7
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
diff	5	2	3	4	7	4	2	3	4	7	4	2	3	4	7
leq	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1
taken	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
ip	1	2	3	3	0	1	2	3	3	0	1	2	3	3	0
writing	0	0	0	0	128	0	0	0	0	128	0	0	0	0	128
cell_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
in	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

3.7 One Word

We agree with whoever is using our computer that the input/output signals will be encoding using the ASCII encoding, meaning if I read 97 it means the latin character “a”, and if I write 98 it means the latin character “b”. 32 is space and 10 is new line. In the ASCII table you can see the last character is 127, so we need only 7 bits to represent all possible characters. Our idealized subleq computer has 64 bit memory cells, meaning we can encode 9 characters 7 bits each, and we still have 1 bit left for the sign. We agree that the value 0 is end of input or end of output. If we read 0 means the input terminates, if we write 0 means the output terminates.

There is a small annoyance that we get the negated input and we write negated value because how how our SUBLEQ works, but we can deal with that.

The code to fetch a word into a memory cell is quite small:

```
macro dbl b
    add b b
end

macro pack c n
    dbl n
    dbl n
    dbl n
    dbl n
    dbl n
    dbl n
    dbl n
    add c n
end

diff: 0
macro ifle a b dest
    mov a diff
    sub b diff
    jle diff dest
end

start:
    jmp do_fetch_word

char: 0
name: 0
blank: 32

do_fetch_word:
    zero name
wgetch:
    zero char
    -1 char
    neg char
    jle char wdone
    ifle char blank wgetch
wtake:
```

```
    pack char name
    zero char
    -1 char
    neg char
    ifle char blank wdone
    jmp wtake
wdone:
    halt
```

The more interesting part here is the `pack` macro, so lets walk through it, imagine we have the word 'hello', or in ascii `104, 101, 108, 108, 111`. We start with `name=0` we double it 7 times, its still 0, then we add 104 to it, after that we get 101, `name=104` we double it 7 times.

```
104 + 104 = 208
    + 208 = 416
    + 416 = 832
    + 832 = 1664
    + 1664 = 3328
    + 3328 = 6656
    + 6656 = 13312
```

Each double moves the data one bit to the left, lets look at it in binary:

```
104   = 1101000
208   = 1101000 0
416   = 1101000 00
832   = 1101000 000
1664  = 1101000 0000
3328  = 1101000 00000
6656  = 1101000 000000
13312 = 1101000 0000000
```

Once we have doubled 7 times we have 7 free bits to put uthe new character, so we put there 101 with addition. and keep going, if the word has more than 9 characters we have a problem we will just kick it out and might corrupt our sign bit, but that is ok for our purposes.

3.8 Dictionary

So we have a word, now we want to build a language, and a language exists in the relationships between words. We need to be able to find a word and its meaning in a list of words and then evaluate it. For that we must have a dictionary somewhere in memory, and we need a way to search for specific word in this dictionary. There are many many ways to do this, but the simplest one is for each added word to have a link to the previous one, and we just need to know where is the last word.

```
latest: 1062
dictionary:   // e.g addr 1000
1000  28130883183 // hello <-.
1008  0                      |
1016  32178484836 // world <-+--.
1024  1000 >-----------------'  |
1032  1718263     // how   <---+--.
1040  1016 >--------------------'  |
1048  1603941     // are   <-------+--.
1056  1032 >-----------------------'  |
1062  1996789     // you              |
1080  1056 >--------------------------'
```

To walk the dictionary looking for a word we simply start from the last word, and jump until we find what we are looking for.

```
diff: 0
macro ifeq a b dest
    mov a diff
    sub b diff
    jle diff .low
    jmp .after
.low:
    zero back
    sub diff back
    jle back dest
.after:
```

```
end

needle: 1718263 // how
link: 0
name: 0
found: 0
do_find_word:
  mov latest link
  zero found
fnext:
  jle link fdone
  load link name // load: mem[name] = mem[mem[link]]
  ifeq name needle ffound
  add one link
  load link link  // follow: mem[link] = mem[mem[link]]
  jmp fnext
ffound:
  mov link found
fdone:
  halt
```

The interesting part here is the ifeq macro and following the link, the example is very small, and I will leave it for you to figure it out with pen and paper.

3.9 Jumping Around

We have two pieces of code: do_find_word and do_fetch_word, we want to be able to jump to a piece of code let it do its thing and then go back to where we were so we can use its result.

Something like this:

```
call do_fetch_word // name contains the word
call do_find_word  // searches for "name" and fills "found"
jle found word_not_found

// do something with the word
```

```
word_not_found:
  halt
```

`?` tells us where we are, we just need to store that somewhere and then use it when we want to go back. A very nice solution to this is a stack of cards, every time we go somewhere we put the address where we want to go back to on top of the deck, when we want to go back we pop the top card and we see where we should jump to. This way if we have to jump to some code which also jumps to some code each step just adds a card on the deck, and pops it back.

```
one: 1
rp: 20000

macro rpush src
    add one rp
    store rp src
end

macro rpop dst
    load rp dst
    sub one rp
end

macro call routine
    rpush .slot
    Z
    Z
    routine
.slot: .back
.back: // we want to continue here
end

macro return
    rpop .ret // fetch from the stack
    Z         // and write to .ret
    Z         // Z Z .ret
.ret: 0
```

```
end

call do_fetch_word
call do_find_word
halt
```

One by one, first lets look at how `call do_fetch_word` macro expands, imagine we are at address 3000:

```
3000 add one rp   // rp += 1
3003 store rp 3043 // mem[mem[rp]] = mem[3043]
3040 Z Z do_fetch_word // jump to do_fetch_word
3043: 3044
3044: call do_find_word
30??: halt
```

So, we first increment where rp points to, then we store the value of where we would like to go back to.

Now imagine in the end of do_fetch_word instead of `halt` we put `return`, and it happens to be on address 2000:

```
...
2000: load rp 2045 // mem[2045] = mem[mem[rp]]
2040: sub one rp
2043: Z Z
2045: 0
```

First we load whatever we stored to go back to and modify the jump instruction to go there. Effectively when we return we go back to where we wanted/

I did not expand `store` and `load` because it will just make things seem more complicated than they are. I added '40' instructions jump which is incorrect when you expand the movs and so on, but it does not matter, its ok to just take a piece of code, inspect it in isolation, and then trust it. We have to learn to see the forest not just the trees or we will waste our life.

```
rp: 20001             // top card of the deck, after the push

caller:
3000  add one rp      // rp: 20000 -> 20001
3003  store rp 3043   // mem[20001] = mem[3043] = 3044
3040  3 3 1900        // jmp do_fetch_word >-------------.
3043  3044            // .slot: the card    >--------.   |
3044  ...             // .back: we continue here <---+   |
                                                     |   |
do_fetch_word:                                       |   |
1900  zero name  <------------------------------------+---'
 ...                                                 |
2000  load rp 2045    // mem[2045] = mem[20001]      |
2040  sub one rp      // rp: 20001 -> 20000          |
2043  3 3             // Z Z, and C is the cell below|
2045  3044            // .ret: the card, again  >----+
                                                     |
return stack:                                        |
20000 0               // where rp began              |
20001 3044            // the card, on the deck  >----'
```

Ok, lets write some imaginary program that has one code calling into another into another into another.

```
one: 1
rp:  20000

macro rpush src
  add one rp
  store rp src
end

macro rpop dst
  load rp dst
  sub one rp
end
```

```
macro call routine
  rpush .slot
  Z
  Z
  routine
.slot: .back
.back:
end

macro return
    rpop .ret
    Z
    Z
.ret: 0
end

na:  -97
nb:  -98
nc:  -99
nnl: -10

start:
  call do_a
  nnl -1
  halt

do_a:
  na -1                  // a, going down
  call do_b
  na -1                  // a, coming back up
  return

do_b:
  nb -1                  // b, going down
  call do_c
  nb -1                  // b, coming back up
  return

do_c:
  nc -1                  // c
```

```
  return
```

When macro expanded it looks like this:

```
0000     3    3   11  Z Z start  // jmp start
0003     0            Z: 0
0004     0            t: 0
0005     1            one: 1
0006 20000            rp:  20000
0007   -97            na:  -97
0008   -98            nb:  -98
0009   -99            nc:  -99
0010   -10            nnl: -10
0011     5    3   14  one Z  // call do_a
0014     3    6   17  Z rp
0017     3    3   20  Z Z
0020    62   62   23  .zap__16 .zap__16  // store rp .slot__2
0023     6    3   26  rp Z  // add rp .zap__16
0026     3   62   29  Z .zap__16
0029     3    3   32  Z Z
0032    63   63   35  .zap1__16 .zap1__16  // mov rp .zap1__16
0035     6    3   38  rp Z  // add rp .zap1__16
0038     3   63   41  Z .zap1__16
0041     3    3   44  Z Z
0044    66   66   47  .put1__16 .put1__16  // mov rp .put1__16
0047     6    3   50  rp Z  // add rp .put1__16
0050     3   66   53  Z .put1__16
0053     3    3   56  Z Z
0056     4    4   59  t t  // zero t
0059    71    4   62  .slot__2 t  // sub .slot__2 t
0062     0            .zap__16: 0
0063     0            .zap1__16: 0
0064    65            ?+1
0065     4            .put__16: t
0066     0            .put1__16: 0
0067    68            ?+1
0068     3            Z
0069     3            Z
0070    78            do_a
```

```
0071    72             .slot__2: .back__2
0072    10   -1   75   nnl -1
0075     3    3   75   Z Z ?  // halt
0078     7   -1   81   na -1  // a, going down
0081     5    3   84   one Z  // call do_b
0084     3    6   87   Z rp
0087     3    3   90   Z Z
0090   132  132   93   .zap__18 .zap__18  // store rp .slot__4
0093     6    3   96   rp Z  // add rp .zap__18
0096     3  132   99   Z .zap__18
0099     3    3  102   Z Z
0102   133  133  105   .zap1__18 .zap1__18  // mov rp .zap1__18
0105     6    3  108   rp Z  // add rp .zap1__18
0108     3  133  111   Z .zap1__18
0111     3    3  114   Z Z
0114   136  136  117   .put1__18 .put1__18  // mov rp .put1__18
0117     6    3  120   rp Z  // add rp .put1__18
0120     3  136  123   Z .put1__18
0123     3    3  126   Z Z
0126     4    4  129   t t  // zero t
0129   141    4  132   .slot__4 t  // sub .slot__4 t
0132     0             .zap__18: 0
0133     0             .zap1__18: 0
0134   135             ?+1
0135     4             .put__18: t
0136     0             .put1__18: 0
0137   138             ?+1
0138     3             Z
0139     3             Z
0140   175             do_b
0141   142             .slot__4: .back__4
0142     7   -1  145   na -1 // a, coming back up
0145   160  160  148   .site__19 .site__19  // return
0148     6    3  151   rp Z  // add rp .site__19
0151     3  160  154   Z .site__19
0154     3    3  157   Z Z
0157     4    4  160   t t  // zero t
0160     0    4  163   .site__19: 0 t
0163   174  174  166   .ret__5 .ret__5  // zero .ret__5
0166     4  174  169   t .ret__5  // sub t .ret__5
```

```
0169     5    6  172  one rp  // sub one rp
0172     3            Z
0173     3            Z
0174     0            .ret__5: 0
0175     8   -1  178  nb -1  // b, going down
0178     5    3  181  one Z  // call do_c
0181     3    6  184  Z rp
0184     3    3  187  Z Z
0187   229  229  190  .zap__22 .zap__22  // store rp .slot__6
0190     6    3  193  rp Z  // add rp .zap__22
0193     3  229  196  Z .zap__22
0196     3    3  199  Z Z
0199   230  230  202  .zap1__22 .zap1__22  // mov rp .zap1__22
0202     6    3  205  rp Z  // add rp .zap1__22
0205     3  230  208  Z .zap1__22
0208     3    3  211  Z Z
0211   233  233  214  .put1__22 .put1__22  // mov rp .put1__22
0214     6    3  217  rp Z  // add rp .put1__22
0217     3  233  220  Z .put1__22
0220     3    3  223  Z Z
0223     4    4  226  t t  // zero t
0226   238    4  229  .slot__6 t  // sub .slot__6 t
0229     0            .zap__22: 0
0230     0            .zap1__22: 0
0231   232            ?+1
0232     4            .put__22: t
0233     0            .put1__22: 0
0234   235            ?+1
0235     3            Z
0236     3            Z
0237   272            do_c
0238   239            .slot__6: .back__6
0239     8   -1  242  nb -1 // b, coming back up
0242   257  257  245  .site__23 .site__23  // return
0245     6    3  248  rp Z  // add rp .site__23
0248     3  257  251  Z .site__23
0251     3    3  254  Z Z
0254     4    4  257  t t  // zero t
0257     0    4  260  .site__23: 0 t
0260   271  271  263  .ret__7 .ret__7  // zero .ret__7
```

```
0263     4  271  266  t .ret__7  // sub t .ret__7
0266     5    6  269  one rp  // sub one rp
0269     3            Z
0270     3            Z
0271     0            .ret__7: 0
0272     9   -1  275  nc -1  // c
0275   290  290  278  .site__25 .site__25  // return
0278     6    3  281  rp Z  // add rp .site__25
0281     3  290  284  Z .site__25
0284     3    3  287  Z Z
0287     4    4  290  t t  // zero t
0290     0    4  293  .site__25: 0 t
0293   304  304  296  .ret__8 .ret__8 // zero .ret__8
0296     4  304  299  t .ret__8  // sub t .ret__8
0299     5    6  302  one rp  // sub one rp
0302     3            Z
0303     3            Z
0304     0            .ret__8: 0
```

Quite intense, but you can think of it like this:

```
start:
0011  call do_a  // deals card 72, jumps to 0078
0071  72         // .slot__2: the card, minted at assembly
0072  nnl -1     // .back__2  <-----------------.
0075  halt                                     |
                                               |
do_a:                                          |
0078  na -1      // a, going down              |
0081  call do_b  // deals card 142             |
0141  142        // .slot__4: the card         |
0142  na -1      // .back__4  <----------------+--.
0145  return     // pops card 72               |  |
                                               |  |
do_b:                                          |  |
0175  nb -1      // b, going down              |  |
0178  call do_c  // deals card 239             |  |
0238  239        // .slot__6: the card         |  |
0239  nb -1      // .back__6  <----------------+-+--.
```

```
0242  return     // pops card 142              |  |  |
                                               |  |  |
do_c:                                          |  |  |
0272  nc -1       // c, the bottom of the well  |  |  |
0275  return     // pops card 239               |  |  |
                                               |  |  |
the deck:                                      |  |  |
20000 0          // where rp began             |  |  |
20001 72         // card  >--------------------'  |  |
20002 142        // card  >-------------------------'  |
20003 239        // card  >----------------------------' <- rp
```

Stacks are also useful to share data between chunks of code, or even within a piece code itself.

```
sp: 300000

macro push src
    add one sp
    store sp src
end

macro pop dst
  load sp dst
  sub one sp
end
```

So we have two stacks, one for data, pointed by `sp`, also called `stack pointer`, and one for returning, pointed by `rp`, sometimes people use `rsp` or `return stack pointer`.

3.10 Self Modification

We can fetch words and find words, and jump around, that is almost enough to bootstrap our language, the only piece we are missing is the ability to modify the dictionary itself.