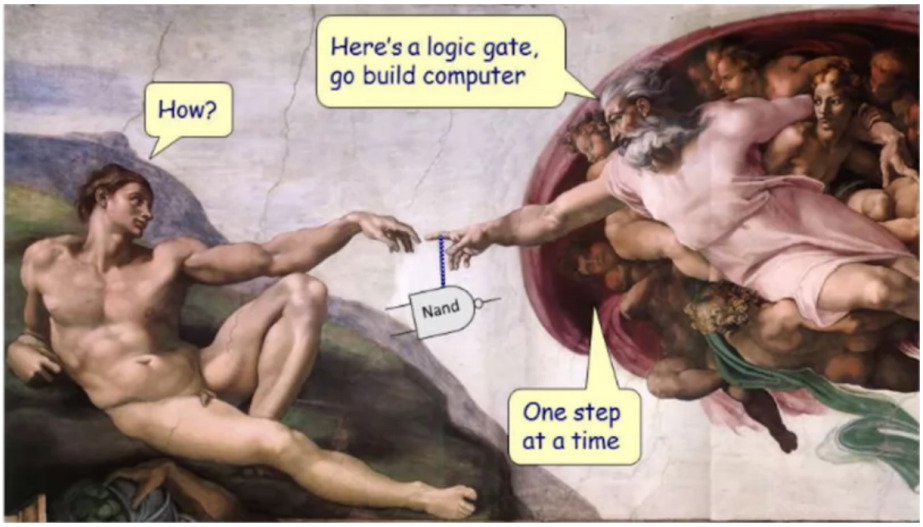
Nand To Tetris

Kunwar Shaanjeet Singh Grover



Contents

1 Introduction **3**
1.1 The Big Picture 3

2 Boolean Functions and Gate Logic **5**
2.1 Boolean Logic 5
2.2 Gate Logic 6
2.3 Hardware Description Language 6
2.4 Multi-Bit Buses 7
2.5 Prespectives 8

3 Boolean Arithematic and The ALU **9**
3.1 ALU: Arithemetic Logic Unit 9
3.2 Prespectives 9

4 Memory **11**
4.1 Combinatorial Logic vs Sequential Logic 11

Chapter 1

Introduction

1.1 The Big Picture

In Part-1 we will build hardware of the computer. In Part-2 we will complete the picture and build the software heirarchy of the computer.

1.1.1 The Road Ahead

How do you actually print "Hello World"? Not writing code for it, but how does it actually work? Why don't we have to worry about it? We only care about "what" is to be done.

"How" ← Implementation "What" ← Abstraction

But who will worry about the "how"? Someone has to do it. A nice thing about computers is once we have done the "how" we only need to worry about the "what".

1.1.2 Multiple Layers of Abstraction

Once we have built the lower level, we dont need to worry about it and can abstract it.

Every week, we will worry about a single level, take the lower level as given, implement the higher level and test that it works.

By the end of the course, we will have built a complete functioning computer and can run anything including games like Tetris.

1.1.3 Two Parts

1. Part-I : Hardware
   (a) Start with *Nand*
   (b) Create the *HACK* computer
2. Part-II : Software
   (a) Start with the *HACK* computer
   (b) Create a fill sotware hierarchy that ...
   (c) ... runs applications like *Tetris*

1.1.4 From Nand To Hack

Nand $\xrightarrow{\text{Combinational Logic}}$ Elementary Logic Gates $\xrightarrow{\text{Comb. and Seq. Logic}}$ CPU, RAM, chipset $\xrightarrow{\text{Digital Design}}$ Computer Architecture $\xrightarrow{\text{Assembler}}$ Low Level Code

1.1.5 How to build a chip

We will use build our chip on a hardware simulator. We will do this in a HDL (Harware Discription Language).

Chapter 2

Boolean Functions and Gate Logic

2.1 Boolean Logic

2.1.1 Boolean Functions Synthesis

Given a Truth Table, how do we construct a boolean function for it?

Lets take an example,

Table 2.1: Truth Table for a Boolean Function

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

This table can be represented by taking OR of the "true"" statements. We can represent the "true" statements by a boolean expression. For example, $x = 0, y = 0, z = 1, f = 1$ can be represented by $\neg x \wedge \neg y \wedge z$. This expression is only true when $x = 0, y = 0, z = 1$ and false on all other cases. So, we can represent every boolean function by AND of such terms.

The truth table 2.1 can be represented by the expression:

$$(\neg x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z)$$

But what is the minimum size expression we can build from a truth table? This is a NP-Complete problem and cannot be solved in polynomial time if $P \neq NP$.

2.1.2 Why NAND?

We can represent every boolean expression only using NAND. This can be trivially proved.

2.2 Gate Logic

A technique for implementing Boolean functions using logic gates.

Logic Gates:

1. Elementary (Nand, And, Or, Not)
2. Composite (Mux, Adder)

2.3 Hardware Description Language

Here is a possible implementation of a XOR gate in HDL.

```
// Xor gate: out = (a And Not(b)) Or (Not(a) And b)

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not (in=a, out=nota);
    Not (int=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or (a=aAndNotb, b=notaAndb, out=out);
}
```

2.3.1 Some comments on HDL

- HDL is a functional / declarative language
- The order of HDL statements is insignificant

- Before using a chip part, you must know its interface. For example:

```
Not(in= ,out= ), And(a=, b= ,out= )
```

- Connections like

```
partName(a=a,...) partName(..., out=out)
```

  are common

Common HDLs:

- VHDL
- Verilog
- Many more HDLs

Our HDL:

- Similar in spirit to other HDLs
- Minimal and simple
- Provides all you need for this course

2.4 Multi-Bit Buses

Sometimes we manipulate "together" an array of bits. It is conceptually convenient to think about such a group of bits as a single entity, sometimes termed "bus". HDLs will usually provide some convenient notion for handling these buses.

2.4.1 Examples on buses in HDL

Here is and example of how buses work in HDL:

```
/*
* Adds three 16-bit values
*/

CHIP Add3Way16 {
    IN first[16], second[16], third[16];
    OUT out[16];

    PARTS:

    Add16(a=first, b=second, out=temp);
    Add16(a=temp, b=third, out=out);
}
```

2.4.2 Sub-buses

Buses can be composed from (and broken into) sub-buses:

```
...
IN lsb[8], msb[8], ...
...
Add16(a[0..7]=lsb, a[8..15]=msb, b=..., out=...);
Add16(..., out[0..3]=t1m out[4..15]=t2);
```

Some syntactic choices of our HDL:

- Overlaps of sub-buses are allowed on output buses of parts
- Width of internal pins is deduced automatically
- "false" and "true" may be used as buses of any width

2.5 Prespectives

In this chapter we built a basic chipset of 15 chips and we will use these to build more advanced chips in later chapters.

We use NAND gates in industry because NAND gates are cheap to make. But since we are looking at lower levels and then creating abstractions, lets see how a NAND gate is made.

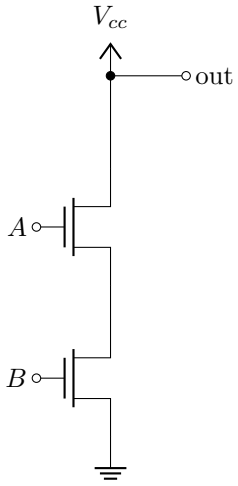


Figure 2.1: A NMOS Nand Gate

If A and B, both are ON, then the current goes from V_{cc} to Ground, and thus $out = 0$ or it goes to out, implying $out = 1$.

Chapter 3

Boolean Arithematic and The ALU

3.1 ALU: Arithemetic Logic Unit

The ALU computes a function of the two inputs, and outputs the result.

Which operations should the ALU perform? This is a hardware/software tradeoff.

3.2 Prespectives

Our ALU is extremly simplified with respect to the industrial level ALUs. This is because the course emphasises simplicity.

Operations like multiplications, division will be left to the software part of course. The tradeoff is that it will be slower but more cost effective.

The 16-bit adder we make is actually not that efficent, because we add the first bits, then compute the carry and then go to the next. This takes a lot of time. A more efficent way is to create lookup table and use it to computer carries faster: Carry Lookahead Adders

Chapter 4

Memory

4.1 Combinatorial Logic vs Sequential Logic

Combinatorial : $out[t] = f(in[t])$

Sequential : $out[t] = f(in[t-1])$