2020-04-05

Build a Modern Computer from First Principles:
From Nand to Tetris

Unit 1.1 - Boolean Logic

e.g. x AND y (x ∧ y)

- Boolean: T or F, 0 or 1
- can make boolean functions out of boolean expressions
  ↳ use truth table to resolve
- Can find **Boolean Identities**
  e.g. commutative law, associative law, distributive law, De Morgan Law

truth table →

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

Unit 1.2 - Boolean Functions Synthesis

Q: How to form Boolean function based on truth table?

x	y	z	f	
0	0	0	1	← (NOT(x) & NOT(y) & NOT(z)) OR
0	0	1	0	
0	1	0	1	← (NOT(x) & y & NOT(z)) OR
0	1	1	0	
1	0	0	1	← (x & NOT(y) & NOT(z))
1	0	1	0	
1	1	0	0	
1	1	1	0	

↳ **disjunctive normal form formula**
- find expressions for each '1' in table that is only true for that row
- combine with 'OR'

Theorem: Any Boolean function can be represented using an expression containing AND, OR and NOT operations

↳ one step further, don't need OR operator

Proof: De Morgan's Law ↳ (X OR Y) = NOT(NOT(x) AND NOT(y))

NAND operator ★

- (x NAND y) = NOT(x AND y) → negation of x AND y

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

Theorem: Any Boolean function can be represented using an expression containing only NAND operations

Proof: 1) NOT(x) = (x NAND x)

2) (x AND y) = NOT(x NAND y)

2020-04-05

Unit 1.3 - Logic Gates

Gate Logic: A technique for implementing Boolean functions using logic gates

Logic Gates:
- Elementary (Nand, And, Or, NOt, ...)
- Composite (Mux, Adder, ...)

Elementary Gates

NAND

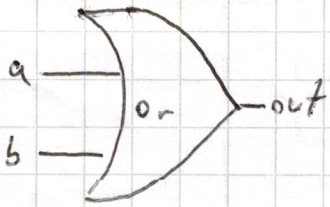
if (a==1 and b==1)
then out=0 else out=1

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

AND

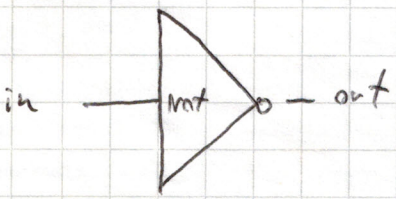
if (a==1 and b==1)
then out=1 else out=0

OR



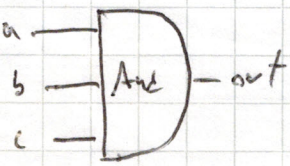
if (a==1 or b==1)
then out=1 else out=0

NOT



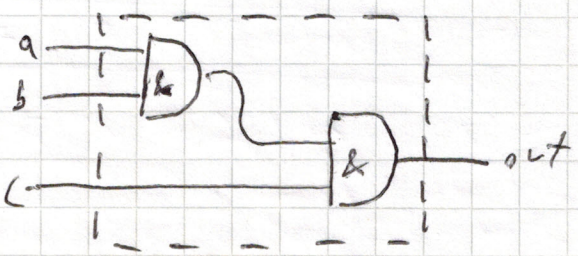
if (in==0)
then out=1 else out=0

Composite Gates



if (a==1 and b==1 and c==1)
then out=1 else out=0

interface
- only one

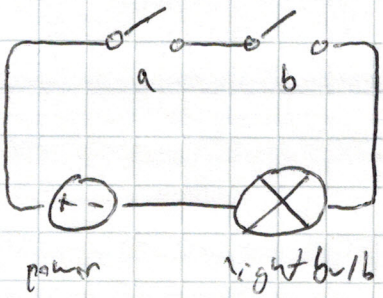


implementation
- possibly many

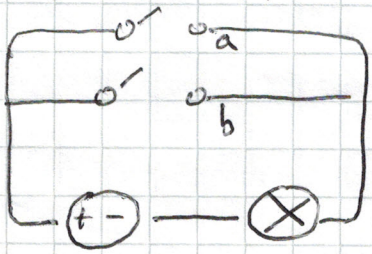
Unit 1.3 - Logic Gates (continued)

Circuit Implementations - circuits that switch on a lightbulb

And circuit



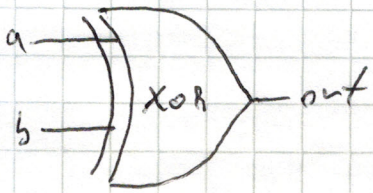
Or circuit



Electrical Engineering

Unit 1.4 - Hardware Description Language

e.g design for following requirement

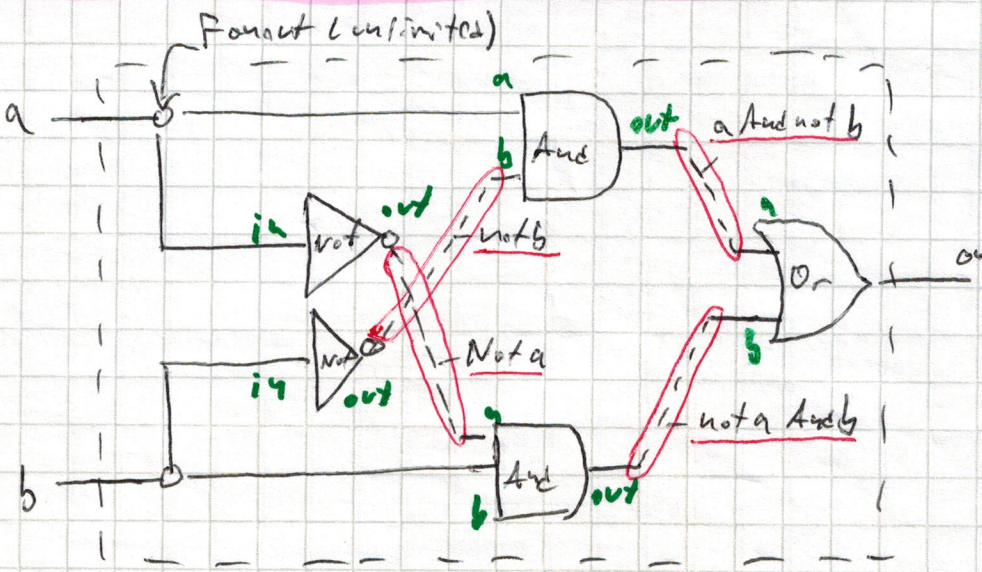


a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

HDL file:

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

```
CHIP Xor {

    IN a, b;
    OUT out;

    PARTS:
    //implementation here
}
```




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

Unit 1.4 - Hardware Description Language (continued)

- HDL is a functional / declarative language
- order of HDL statements is insignificant (but commonly written left to right)
- Before using a chip part, you must know its interface
  e.g. Not(in=, out=), And(a=, b=, out=), Or(a=, b=, out=)
- Connections like partName(a=a, ...) and partName(..., out=out) are common

- Common HDLs: VHDL, Verilog, many others

Unit 1.5 - Hardware Simulation

Simulation options: interactive, script-based, with/without compare file

Simulation Process:
- Load the HDL into hardware simulator
- Enter values (0/1) into chip's input pins (e.g. a and b)
- Evaluate chip's logic
- Inspect resulting values of:
  - Output pins (e.g. out)
  - Internal pins (e.g. nota, notb, a and notb, notaandb)

Behavioral simulation:
- implement chip logic in a high level language (Python, Java) not HDL
- run test script, & then rename output file to compare file

Hardware Construction Projects:
- The players: (first approximation)
  a) System Architects
  b) Developers
- System architects decide which chips are needed
- For each chip, the architect creates:
  a) a chip API (chip name, i/o pins) ~~+ test, compare file~~
  b) a test script
  c) a compare file
- Given these resources, a developer can build the chip (in HDL)

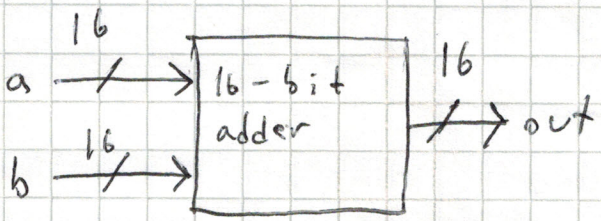
⟹ Developer's Point of View

Given following files
1) stub file (HDL file without PARTS implementation)
2) Test script
3) compare file

Unit 1.6 - Multi-Bit Buses

- Sometimes we manipulate together an array of bits
- Conceptually convenient to think about such a group of bits as a single entity, sometimes termed "bus" ← latin root
- HDLs usually provide convenient notation for handling buses

e.g.



- 32 input wires, 16 output wires

HDL:

```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    ...
}
```

⇒

```
// 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);
}
```

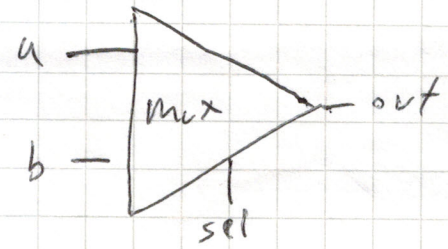
Working with bits in buses:

```
// ANDs together all 4 bits of input. output is a single bit.

CHIP And4Way {
    IN a[4];
    OUT out;

    PARTS:
    AND(a=a[0], b=a[1], out=t01);
    AND(a=t01, b=a[2], out=t012);
    AND(a=t012, b=a[3], out=out);
}
```

- multi-bit busses are indexed right to left
  ↳ If A is a 16-bit bus, then A[0] is rightmost bit, A[15] is leftmost bit

Unit 1.7 – Project Overview

Multiplexor Gate (Mux)



a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

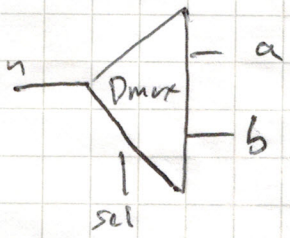
sel	out
0	a
1	b

– abbreviated truth table

```
if (sel==0)
   out=a
else
   out=b
```

- 2-way multiplexor enables selecting, and outputting, one of two possible inputs
- Widely used in: digital design, communication networks
- Can be implemented using And, Or, and Not gates

Demultiplexor (DMux)

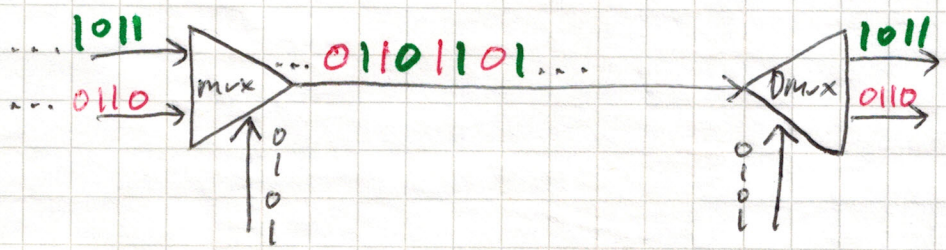



```
if (sel==0)
   {a,b}={in,0}
else
   {a,b}={0,in}
```

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- Acts like 'inverse' of multiplexor
- Distributes single input value into one of two possible values

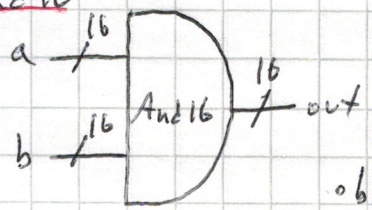
Example: Mux/DMux in communication Networks



- each sel bit is connected to an oscillator that produces a repetitive train of ~~oscillating~~ alternating 0 & 1 signals
- enables transmitting multiple messages on a single, shared communication line
- Can be completely asynchronous (encode/decode)

Unit 1.7 - Project Overview (continued)

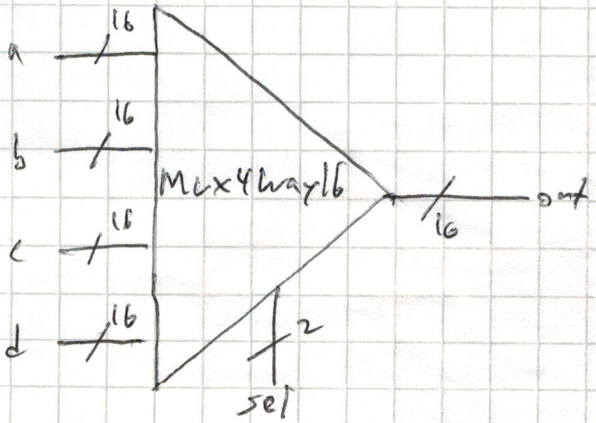
And16




```
a = 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0 0
b = 0 0 1 0 1 1 0 1 0 0 1 0 1 0 1 0
out = 0 0 1 0 1 0 0 1 0 0 0 0 1 0 0 0
```

- bitwise And applied over array, one 16-bit output

16-bit, 4-way multiplexor

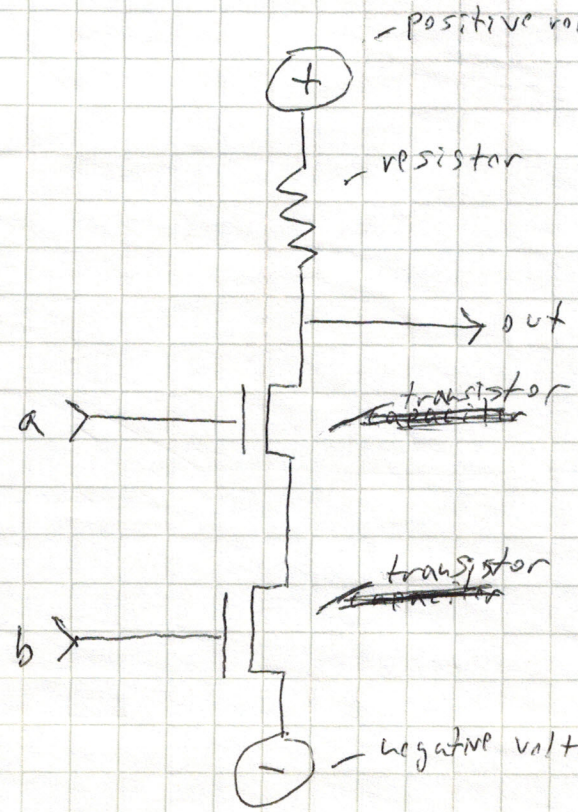


sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

- can be built using several Mux16 gates

Unit 1.8 - Perspectives

Physical Implementation of a NAND gate - NMOS implementation:



How it works:
- if a & b are true, a positive voltage is applied to transistors,
- a full connection formed from negative voltage to output
- in any other case, the positive voltage will rule because no connection made to negative terminal
- the weak connection is overruled by the strong negative connection: if a=1, b=1