Lecture 2

Gidon Rosalki

2025-03-30

1 CMOS and logic

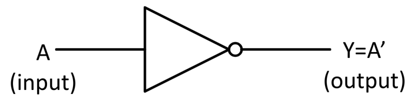
Last semester we discussed what are N-MOS and P-MOS transistors, and how they work. As we recall, providing 0 to the gate (or low voltage), will open the switch, and no current will flow between the source and the drain, whereas if we provide 1, or high voltage, this will close the switch, allowing current to flow from the source to the drain. The exact physics of how this works is now less interesting, and we will instead be discussing the creation of circuits using these capabilities.

1.1 Simple circuits

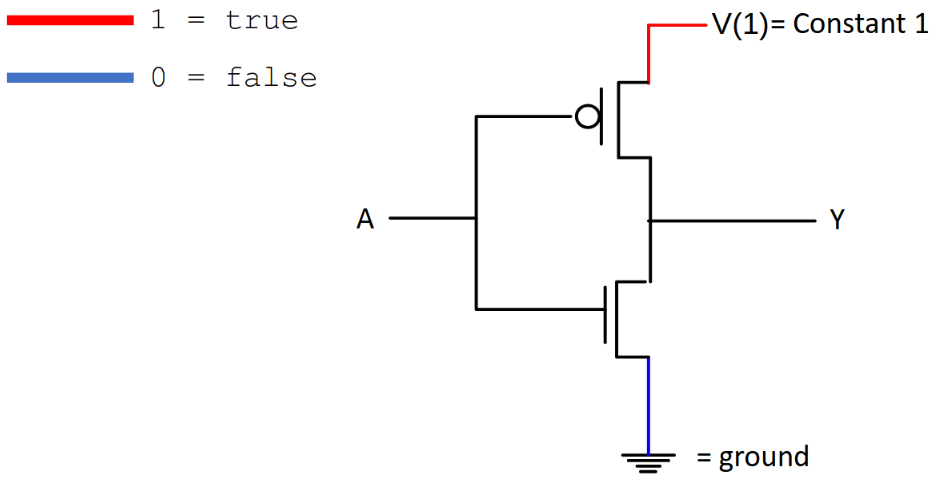
1.1.1 NOT gate - inverter

A	Y
0	1
1	0

Table 1: δ



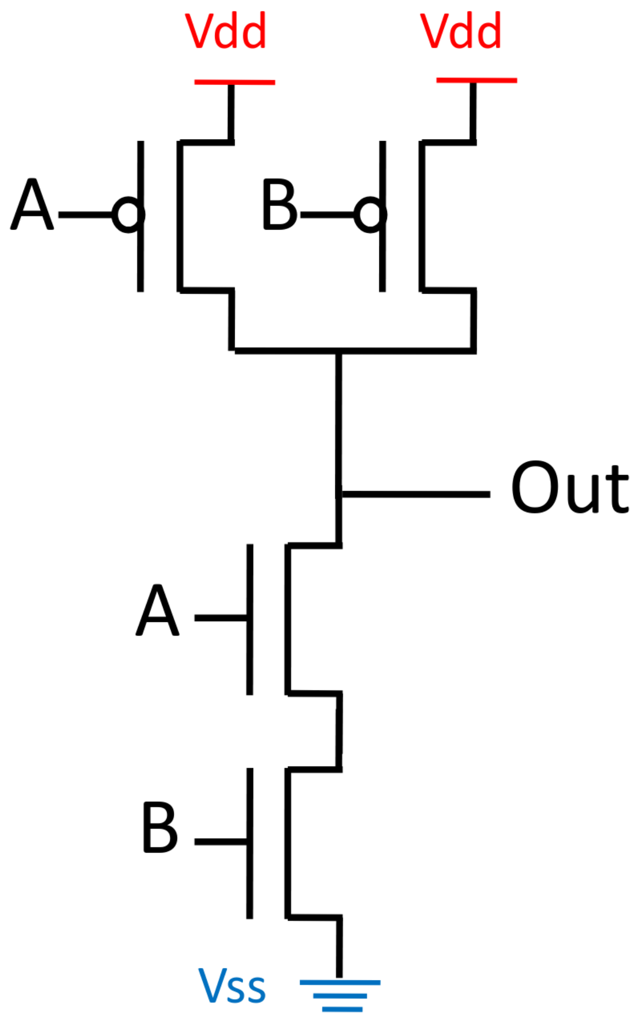
The inverter takes the input voltage, and if it is high returns low, and low returns high. How do we build this we P-MOS and N-MOS?



1.1.2 NAND

A	B	$NAND(A, B)$
0	0	1
0	1	1
1	0	1
1	1	0

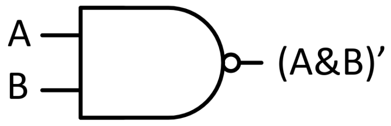
Table 2: δ



1.1.3 Transistor level logic circuits - CMOS

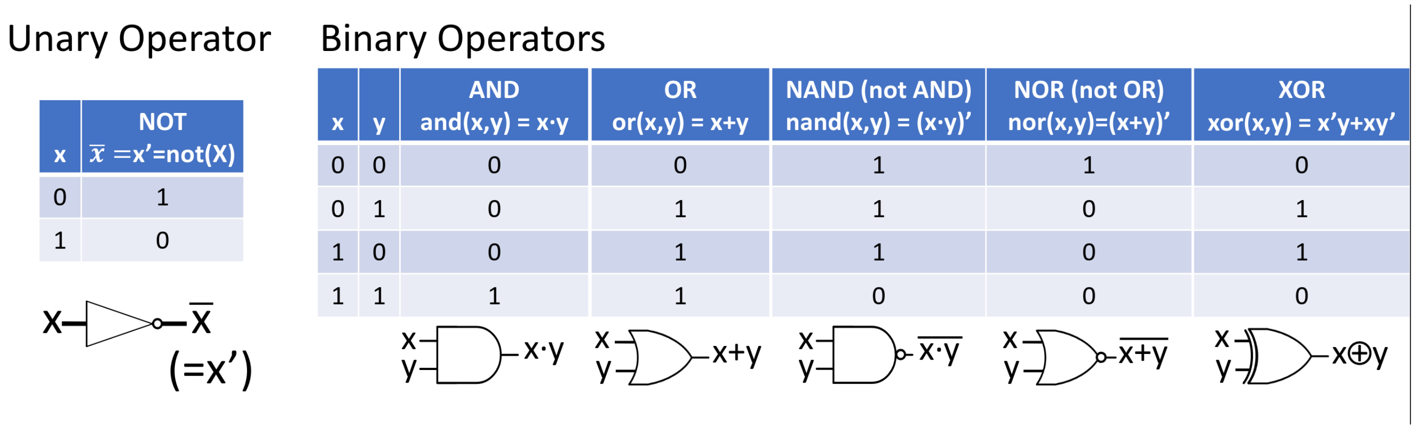
The rules for CMOS (Complementary Metal Oxide Semiconductor) circuits are that PMOS and NMOS are always in pairs, and that PMOS is always to V(1), where NMOS is always towards GND(0).

1.2 Boolean functions



Boolean variables are bi-value objects (bits in computer science). They are often 1 or 0, often also defined as true or false. **Boolean algebra** defines operations on boolean variables, for example AND, OR, NOT are all operations. **Boolean functions** map each variable's assignment to a boolean result. They can also be expressed using variables, operations, and parentheses, or in a truth table. Functions can have n inputs, and m outputs. Each assignment of variables results with a specific output.

1.2.1 Basic boolean functions



Unary Operator

x	NOT $\overline{x}$ =x'=not(X)
0	1
1	0

Binary Operators

x	y	AND and(x,y) = x·y	OR or(x,y) = x+y	NAND (not AND) nand(x,y) = (x·y)'	NOR (not OR) nor(x,y)=(x+y)'	XOR xor(x,y) = x'y+xy'
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

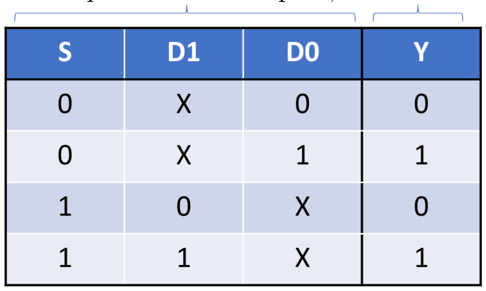
x → $\overline{x}$ (=x')
x, y → $x \cdot y$
x, y → $x+y$
x, y → $\overline{x \cdot y}$
x, y → $\overline{x+y}$
x, y → $x \oplus y$

There are more operators (such as NXOR), but they are generally intuitive to understand what they do based off the name.

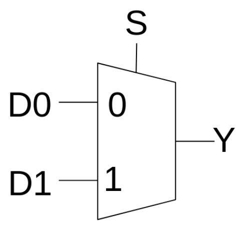
A **minterm** is a product (AND) of all the functions variables, in direct or complemented form. A minterm is equal to 1 on exactly one row of the truth table. A **maxterm** is a sum (OR) of all the functions variables, in direct or complemented form. A maxterm is equal to 0 on exactly one row of the truth table. The Sum of Products (SoP) is the sum of all the minterms that return 1. The Product of Sums (PoS) is the product of all the 0 maxterms. Minimal PoS / SoP is the **most simplified form**.

1.3 Combinatorial logic

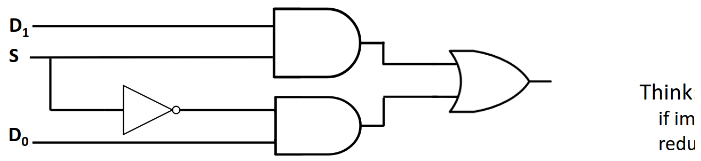
The 2:1 multiplexer takes 2 inputs, and chooses one of them.



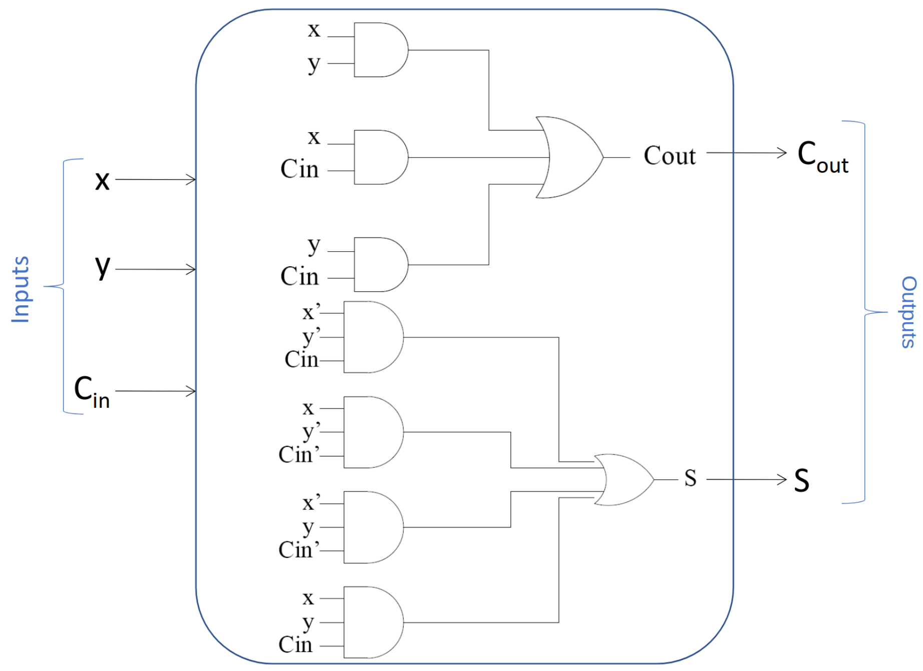
S	D1	D0	Y
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

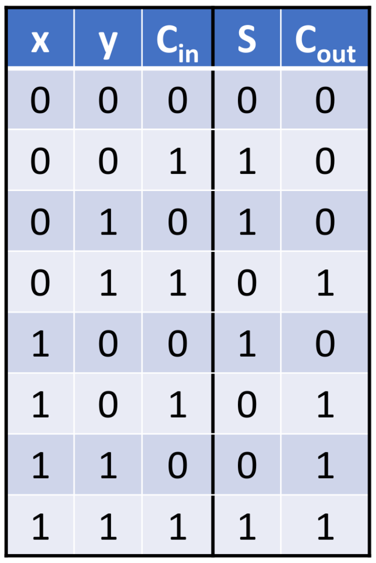


$$MUX(S,D_0,D_1) = S \cdot D_1 + S' \cdot D_0$$



If this were implemented in CMOS, how could we reduce the transistor count?





x	y	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The full adder is the building block for adding two binary numbers. It handles a single bit. It takes the inputs x, y, C_{in}, and outputs S, C_{out}. It calculates $x + y + C_{in}$, placing the sum in S, and the carry in C_{out}.

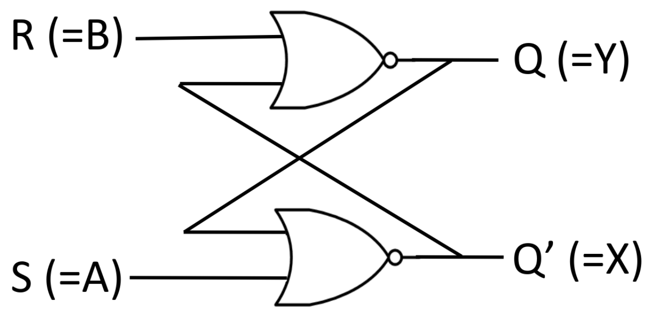
1.3.1 Standard units

The decoder has n inputs, and 2^n outputs. Given the input as a binary number, it will turn on only the relevant number in the output. For example, given the input 11, then it will turn on bit 3 in the output (remember, both start from 0).
The demultiplexer has a 1 bit input i, and an n bit selector, with a 2^n bit output, and according to the value of the selector, it selects to which output to give the input.
The encoder is the inverse of the decoder. It takes a location in 2^n inputs, and then outputs the associated binary number in the n outputs.
Finally the multiplexer has 2^n inputs, and n selector bits, giving a single output f.

1.4 Sequential circuits

These are logical circuits that contain a feedback loop. The outputs are determined by both the inputs, and the value in memory. The value in memory is determined by the previous values of the circuit (inputs, outputs, memory). The order of the values in the inputs matters. Asynchronous sequential circuits have no synchronisation of state change with a clock, and the outputs and state can change at any time. Synchronous sequential circuits have their state changes synchronised to a clock. The clock is a repetitive waveform (0-1-0-1...). The level of the clock can be either 0 or 1, and the edge is either positive/rising or negative/falling.

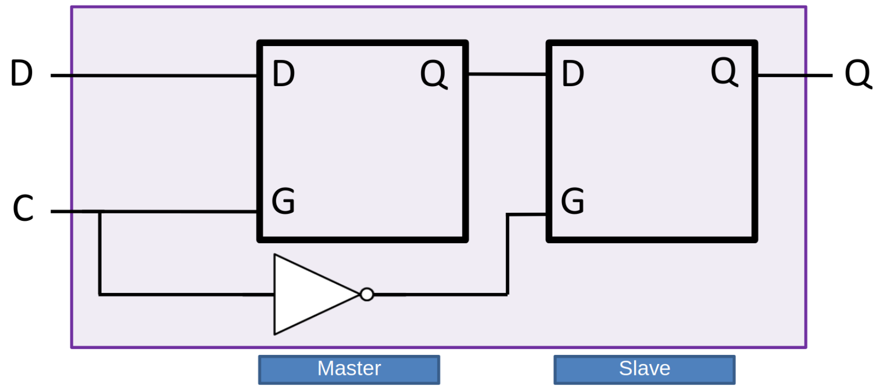
Using this concept we may build a Set Restore (SR) latch, which can be used to store a bit in memory:



We have effective got two inputs, Set and Reset. When $S = 1$ it sets Q to 1. $R = 1$ sets $Q = 0$. Leaving them both as 0 does not change the output, and having them both be 1 is an illegal state.

S	R	Q(t+1)	Q'(t+1)	
0	0	Q(t)	Q'(t)	$\Rightarrow$ Assuming X = Y'
0	1	0	1	
1	0	1	0	
1	1	0	0	$\Rightarrow$ Illegal

A gated SR-latch adds a gate to the inputs, which is combined with each of the inputs individually through AND gates, such that if the gate is 0, then the latch is locked, and the outputs won't change. If the gate is 1, then it operates normally. The gate is often connected to the clock. Since we often do not need to separate the inputs, we can make a gated D-latch, where there are only 2 inputs. The gate, and D, which is given to both R and S, but to get to R it first passes through a not gate. Therefore, when $D = 0$ (and the gate is unlocked), then the memory bit is set to 0. If D is set to 1, then the memory bit is set to 1. Let us suppose we want the output after 3 ticks of the clock. If we just just connect 3 latches in series, then within one tick of the clock it is possible that we will already have output from the last latch. This is because latches operate on the **level** of the clock, rather than the **edge**. Flip flops are edge triggered, rather than level triggered, and are thus sensitive to input changes only around the very short time of the clock rising/falling edge. Here is an example of a D Flip Flop:



Depending on the location of the NOT gate will change whether or not the D Flip Flop activates on the rising or falling edge. The above example is a falling edge flip flop, and if the NOT gate was connected to the first latch instead, then it would be a rising edge flip flop.