Boolean Logic

- AB = (A)(B) = A and B
- A + B = A or B
- A with a bar on top = notA
- To solve these problems, use the chart in discussion and see if the problem has a pattern from the left side of an equation; if it does, then convert it to the thing on the right. The distributive one can work both ways (as in it might also be helpful to go from right to left).
- Not in the chart, but may be helpful:
    - A + notA * B = A + B
        - Example: A + C(notA)B = A + CB
    - Also: (A + B)(C + D) = AC + BC + AD + BD
- When in doubt:
    - 1. think logically
        - For example, (B + C)(notB + notC) means (B or C) and (notB or notC). If it's easier for you to think logically, you might be able to just reason out that this expression simplifies to (B and notC) or (notB and C), which is (B)(notC) + (notB)(C)
    - 2. try writing out a truth table
        - If you have the time to do so, you can also use a truth table. For the example above → plugging into (B + C)(notB + notC):
        -

B	C	Output
0	0	0
0	1	1
1	0	1
1	1	0

        - From this truth table, we can see that this is the same as (notB)(C) + (B)(notC)

States

For this context, when we say register, we mean something that temporarily stores values in circuits. Here is my attempt to build an intuition behind this stuff. Feel free to skip ahead if you just want the formulas and definitions :)

Take this small example: input register ——————

adder —— output register

3 ————

Registers hold values and don't let them move forward until we tell them to. If we want to calculate 6+3, then we would set the input register to 6, then have the register let 6 through into the circuit, then measure the output register.

We want the register to let out values slower than the adder takes to compute (think about why?), so logically the register shouldn't change until the adder is finished.

time between register changes >= time for adder to compute

How do we specify when a register changes? With a clock tick. A clock will specify **constant time intervals** for when register values change. There are upticks and downticks, but we generally only care about upticks. We will allow registers to change on upticks. Therefore,

time between clock upticks >= time for adder to compute

We're going to call this a cycle time (more on this later), so

cycle time >= time for adder to compute

But wait, there's more. In order to work with registers, we have to take into account their downsides as well, since they're not perfect. Registers themselves have delays. We define clock_to_q = time between uptick and the value actually leaving the register. Then,

cycle time >= clock_to_q + time for adder to compute

Also, in order for registers to work, we must keep the input stable for a certain amount of time around when we tell it to change. Before = setup, after = hold. Hold time, however, is included in the clock_to_q and adder time (there's an explanation for this, included later). So…

cycle time >= clock_to_q + time for adder to compute + setup

Finally, to generalize this, we say

cycle time >= clock_to_q + longest CL + setup

CL stands for combinatorial logic. If we have a complicated circuit, longest CL means the longest path we can take from any one register to any other register.

Also, *max hold time <= clock_to_q + shortest CL*

Shortest CL means the shortest path we can take from any one register to any other register. When I say longest/shortest path, I mean the longest time/shortest time.

Formulas

cycle time >= clock_to_q + longest CL + setup
Cycle time is the time it takes for an input to leave any one register and go into any other register. We want to minimize this, since the shorter it is, the faster our circuit is. But it has to be greater than clock_to_q + longest CL + setup. Note: setup time will be the setup time of the *receiving* register, since we are setting up for the next cycle.

frequency = 1/cycle time

max hold time <= clock_to_q + shortest CL
A lot of people asked about this in discussion, so I'll explain the logic here. Remember that hold time is the time *after* the clock uptick that our input to a register has to stay constant, in order for the register to work correctly. *Hold time is a property of the register*. Therefore, what this equation is saying is that we cannot work with a register if its hold time exceeds clock_to_q + shortest CL. If this was the case, then the next cycle's input might finish computing and reach the register *before* the previous cycle's hold time finished. If this were the case, then the input would change *during* the hold time, making the register not work.

Definitions

clock_to_q - time between clock uptick and value actually leaving the register (registers take time to send a value through after the value has been stopped)

setup time - time before clock tick input must be constant (in order for the register to measure correctly)

hold time - time after clock tick input must be constant (in order for the register to measure correctly). Hold time is *included* in clock_to_q + shortest CL, so don't worry about it too much.

cycle time - time it takes for an input to leave one register and go into another. *We* define the cycle time, and generally we try to make it as short as possible because then our circuit runs faster :)