CS61C Summer 2025

# Precheck: Boolean Algebra, SDS

Discussion 7

## 1 Boolean Logic Precheck

1.1 Simplifying boolean logic expressions will not affect the performance of the hardware implementation.

False. Different gate arrangements that implement the same logic can have different propagation delays, which can affect the allowable clock speed.

1.2 The fewer logic gates, the faster the circuit (assuming each gate has the same propagation delays).

False. A wide circuit with more gates in parallel can have less delay than just a few gates arranged in sequence.

1.3 The time it takes for clock-to-q and register setup can be greater than one clock cycle.

False. This can result in instability if registers are connected to each other, as register outputs may not have propagated properly before the next rising edge.

1.4 Every possible combinational logic circuit can be expressed by some combination of NOR gates.

True. NOR can be used to express AND, OR, and NOT gates. Thus, NOR is functionally complete and can be used to represent any possible Boolean expression, and thus any combinational logic circuit.

1.5 The shortest combinational logic path between two state elements is useful in determining circuit frequency and minimum clock cycle.

False. The minimum clock cycle must allow enough time for every combinational logic delay to settle on an output, so the frequency is based on the longest combinational logic delay possible between state elements.

## 2 Boolean Logic

In digital electronics, it is often important to get certain outputs based on your inputs, as laid out by a truth table. Truth tables map directly to Boolean expressions, and Boolean expressions map directly to logic gates. However, in order to minimize the number of logic gates needed to implement a circuit, it is often useful to simplify long Boolean expressions.

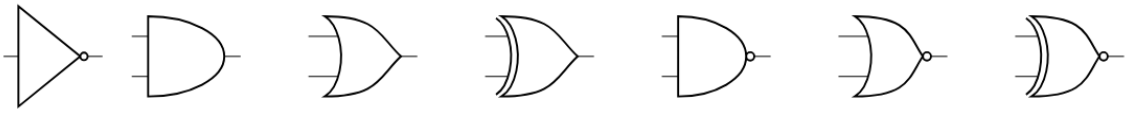
We can simplify expressions using the nine key laws of Boolean algebra:

| Name | AND Form | OR form |
|---|---|---|
| Commutative | $x \cdot y = y \cdot x$ | $x + y = y + x$ |
| Associative | $(xy)z = x(yz)$ | $(x + y) + z = x + (y + z)$ |

| Name | AND Form | OR form |
|---|---|---|
| Identity | $x \cdot 1 = x$ | $x + 0 = x$ |
| Null | $x \cdot 0 = 0$ | $x + 1 = 1$ |
| Absorption | $x \cdot (x + y) = x$ | $x + x \cdot y = x$ |
| Distributive | $(x + y) \cdot (x + z) = x + yz$ | $x \cdot (y + z) = xy + xz$ |
| Idempotent | $x \cdot x = x$ | $x + x = x$ |
| Inverse | $x \cdot \overline{x} = 0$ | $x + \overline{x} = 1$ |
| De Morgan's | $\overline{x \cdot y} = \overline{x} + \overline{y}$ | $\overline{x + y} = \overline{x} \cdot \overline{y}$ |

Additionally, we have many boolean functions which take boolean signals (0 or 1) as input and output a boolean result (0 or 1). When designing digital systems, boolean functions are represented as **logic gates**.

2.1 Label each of the following logic gates with their respective boolean function, and draw a truth table representing their outputs:



NOT, AND, OR, XOR, NAND, NOR, XNOR

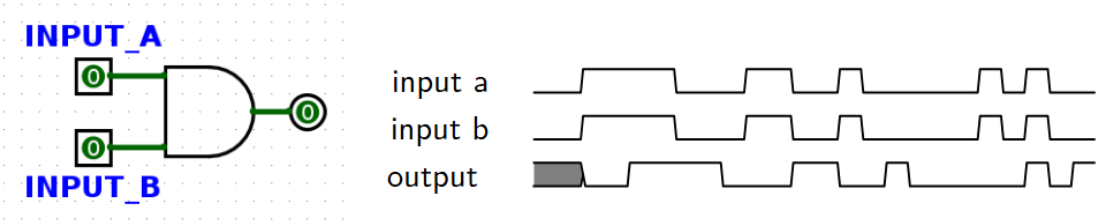
Here are the outputs for each boolean function combined into a single truth table. All possible combinations of the inputs $x$ and $y$ are shown the left, and the output of the the boolean function based on the current inputs is shown on the right.

| **Input(s)** | | **NOT** | **AND** | **OR** | **XOR** | **NAND** | **NOR** | **XNOR** |
|---|---|---|---|---|---|---|---|---|
| $x$ | $y$ | $\overline{x}$ | $x \cdot y$ | $x + y$ | $x \oplus y$ | $\overline{x \cdot y}$ | $\overline{x + y}$ | $\overline{x \oplus y}$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

# 3 SDS

There are two basic types of circuits: combinational logic circuits and state elements.

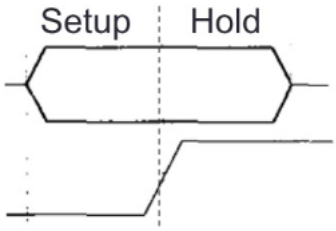
**Combinational logic** circuits simply change based on their inputs after whatever propagation delay is associated with them. For example, if an AND gate (pictured below) has an associated propagation delay of 2ps, its output will change based on its input as follows:




You should notice that the output of this AND gate *always* changes 2ps after its inputs change.

**State elements**, on the other hand, can *remember* their inputs even after the inputs change. State elements change value based on a clock signal. A rising edge-triggered register, for example, samples its input at the rising edge of the clock (when the clock signal goes from 0 to 1).

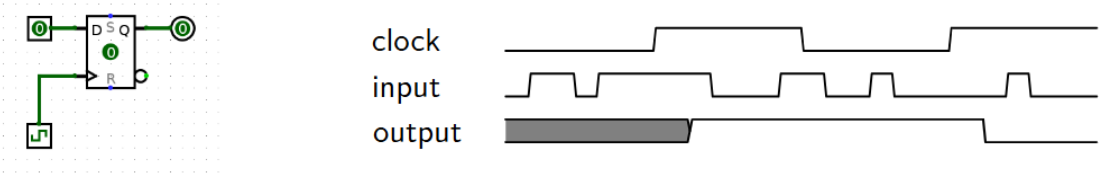
Like logic gates, registers also have a delay associated with them before their output will reflect the input that was sampled. This is called the **clk-to-q** delay. ("Q" often indicates output). This is the time between the rising edge of the clock signal and the time the register's output reflects the input change.




The input to the register samples has to be stable for a certain amount of time around the rising edge of the clock for the input to be sampled accurately. The amount of time before the rising edge the input must be stable is called the **setup** time, and the time after the rising edge the input must be stable is called the **hold** time. Hold time is generally included in clk-to-q delay, so clk-to-q time will usually be greater than or equal to hold time.

Logically, the fact that clk-to-q $\geq$ hold time makes sense since it only takes clk-to-q seconds to copy the value over, so there's no need to have the value fed into the register for any longer.

Examine the register circuit and assume **setup** time of 2.5ps, **hold** time of 1.5ps, and a **clk-to-q** time of 1.5ps. The clock signal has a period of 13ps.




Notice that the value of the output in the diagram doesn't change immediately after the rising edge of the clock. Until enough time has passed for the output to reflect the input, the value held by the output is garbage; this is represented by the shaded gray part of the output graph. Clock cycle time must be small enough that inputs to registers don't change within the hold time and large enough to account for clk-to-q times, setup times, and combinational logic delays.

A few important SDS relationships are below:

$$\tau_{\text{critical path delay}} = \tau_{\text{clk-to-q}} + \tau_{\text{combinational logic delay}} + \tau_{\text{setup time}}$$

where $\tau_{\text{combinational logic delay}}$ is the maximum combinational logic delay for any register $\longrightarrow$ register path in the circuit. The path with the maximum delay is called the "critical path".

Additionally, circuits must satisfy hold-time constraints because hold times may be violated if data propagates too quickly (see above):

$$\tau_{\text{clk-to-q}} + \tau_{\text{smallest combinational delay}} \geq \tau_{\text{hold time}}$$

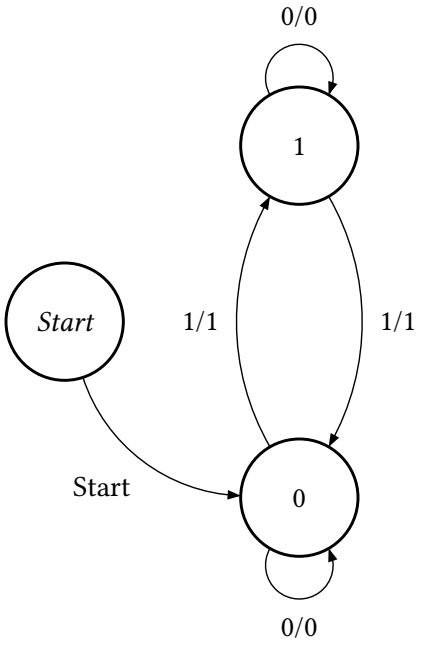
# 4 FSM

A finite state machine is a type of simple automaton where the next state and output depend only on the current state and input. Each state is represented by a circle, and every proper finite state machine has a starting state, signified either with the label "Start" or a single arrow leading into it. Each transition between states is labeled `[input]/[output]`.

For example, below is a finite state machine with two states (0 and 1). It outputs `1` when the state changes, and `0` when the state stays the same.

The machine starts in state `0`. When the input is `0`, it stays in its current state and outputs `0`. When the input is `1`, it switches to the other state and outputs `1`.

When in state `1`, the machine behaves the same way: it stays in state `1` and outputs `0` when the input is `0`, and switches back to state `0` with an output of `1` when the input is `1`.




With combinational logic and registers, any FSM can be implemented in hardware!