CS61C
Summer 2025

Boolean Algebra, SDS

Discussion 7

1 Boolean Logic

1.1 Simplify the following Boolean expressions:

(a) $(A + B)\left(A + \overline{B}\right)C$ = AC

A breakdown of the boolean algebra laws used to simplify the expression is shown below:

$$\begin{aligned}
(A + B)\left(A + \overline{B}\right)C &= \left(AA + A\overline{B} + A\overline{B} + B\overline{B}\right)C && \text{Distributive Property}\\
&= \left(A + A\overline{B} + 0\right)C && \text{Idempotent \& Inverse Properties}\\
&= \left(A + A\overline{B}\right)C && \text{Identity Property}\\
&= A\left(1 + \overline{B}\right)C && \text{Distributive Property}\\
&= AC && \text{Null Property}
\end{aligned}$$

(b) $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,\overline{C} + A\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C + A\,\overline{B}\,C$ = $\overline{C} + A$

$$\begin{aligned}
\overline{A}\,\overline{C}\left(\overline{B} + B\right) + A\,\overline{C}\left(B + \overline{B}\right) + AC\,\left(B + \overline{B}\right) &= \overline{A}\,\overline{C} + A\overline{C} + AC && \text{Distributive}\\
&= \overline{A}\,\overline{C} + A\overline{C} + A\overline{C} + AC && \text{Idempotent}\\
&= \left(\overline{A} + A\right)\overline{C} + A\left(\overline{C} + C\right) && \text{Distributive}\\
&= \overline{C} + A && \text{Inverse}
\end{aligned}$$

Alternatively, to simplify $\overline{A}\,\overline{C} + A\overline{C} + AC$ with the distributive property $x + yz = (x + y) \cdot (x + z)$:

$$\begin{aligned}
\overline{A}\,\overline{C} + A\overline{C} + AC &= \overline{C}\left(\overline{A} + A\right) + AC\\
&= \overline{C} + AC\\
&= \left(\overline{C} + A\right)\left(\overline{C} + C\right) && \text{Distributive (AND form)}\\
&= \overline{C} + A
\end{aligned}$$

(c) $\overline{A\left(\overline{B}\,\overline{C} + BC\right)}$ = $\overline{A} + B\overline{C} + \overline{B}C$

$$\begin{aligned}
\overline{A\left(\overline{B}\,\overline{C} + BC\right)} &= \overline{A} + \overline{\overline{B}\,\overline{C} + BC} && \text{De Morgan's}\\
&= \overline{A} + \overline{\left(\overline{B}\,\overline{C}\right)}\,\overline{BC} && \text{De Morgan's}\\
&= \overline{A} + (B + C)\left(\overline{B} + \overline{C}\right) && \text{De Morgan's}\\
&= \overline{A} + B\overline{C} + \overline{B}C && \text{Distributive}
\end{aligned}$$

(d) $\overline{A}(A+B)+(B+AA)\left(A+\overline{B}\right)$ = $A+B$

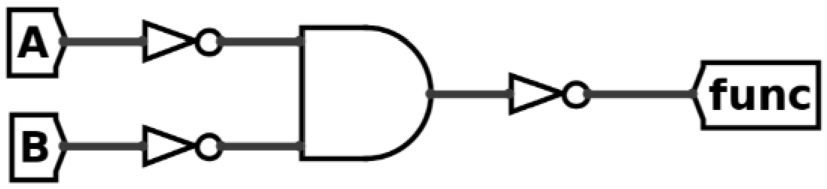
$$\begin{aligned}
\overline{A}(A+B)+(B+AA)\left(A+\overline{B}\right) &= \overline{A}(A+B)+(A+B)\left(A+\overline{B}\right) && \text{Idempotent}\\
&= (A+B)\left(\overline{A}+A+\overline{B}\right) && \text{Distributive}\\
&= (A+B)\left(1+\overline{B}\right) && \text{Inverse}\\
&= (A+B) && \text{Null}
\end{aligned}$$

2 Digital Logic Simplification

For the following digital logic circuits:

1. Write a boolean algebra expression that corresponds the physical circuit.
2. Simplify the expression and draw the simplified circuit.

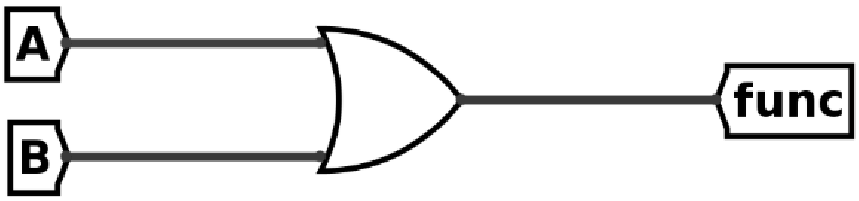
2.1



We can start by labeling the inputs / outputs of each of our logic gates. The first two gates after the A and B inputs are NOT gates which output $\overline{A}$ and $\overline{B}$ respectively. These are fed as inputs to the AND gate which will output $\overline{A} \cdot \overline{B}$. Lastly, this is fed into NOT gate which is our output expression $F(A, B) = \overline{\overline{A} \cdot \overline{B}}$. We can simplify this expression with De Morgan's law to get a simplified expression:

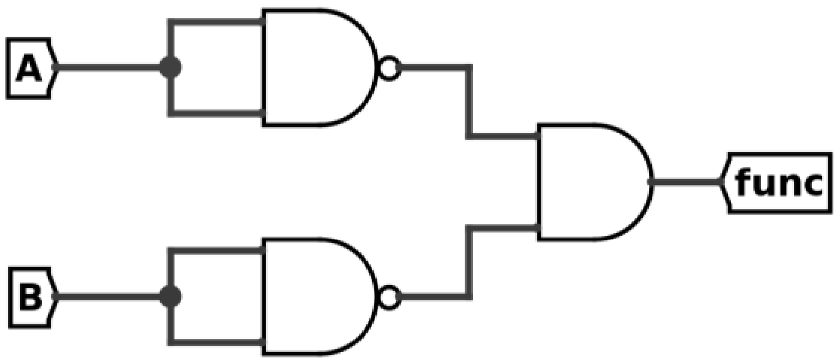
$$\begin{aligned} F(A, B) &= \overline{\overline{A} \cdot \overline{B}} \\ &= \overline{\overline{A}} + \overline{\overline{B}} \\ &= A + B \end{aligned}$$

Redrawing our simplified circuit, we get:



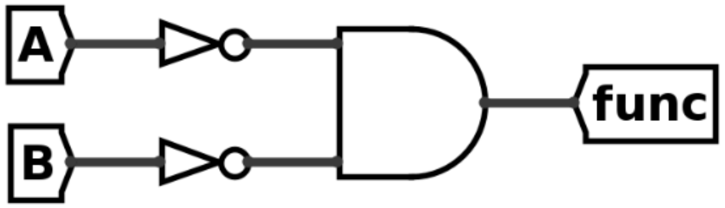
Which is just an OR gate. As extra practice, you can verify the simplification by writing truth tables for each expressions and verifying that they match.

$\boxed{2.2}$

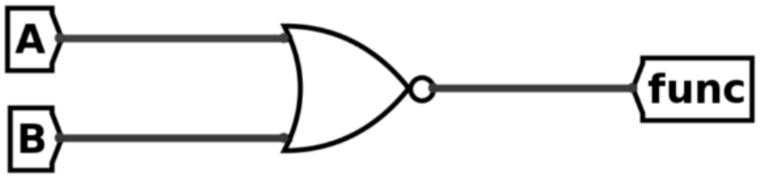


Following similar logic as above, can write expressions for the inputs and outputs of each of the logic gates. The first NAND gate has inputs A and A for an output of $\overline{\mathrm{A} \cdot \mathrm{A}}$ ("not A and A") and similarly for the second NAND gate. Lastly, the outputs are fed into an AND gate and can be simplified:

$$\begin{aligned} \mathrm{F}(\mathrm{A}, \mathrm{B}) &= \left(\overline{\mathrm{A} \cdot \mathrm{A}}\right)\left(\overline{\mathrm{B} \cdot \mathrm{B}}\right) \\ &= \overline{\mathrm{A}} \cdot \overline{\mathrm{B}} \end{aligned}$$

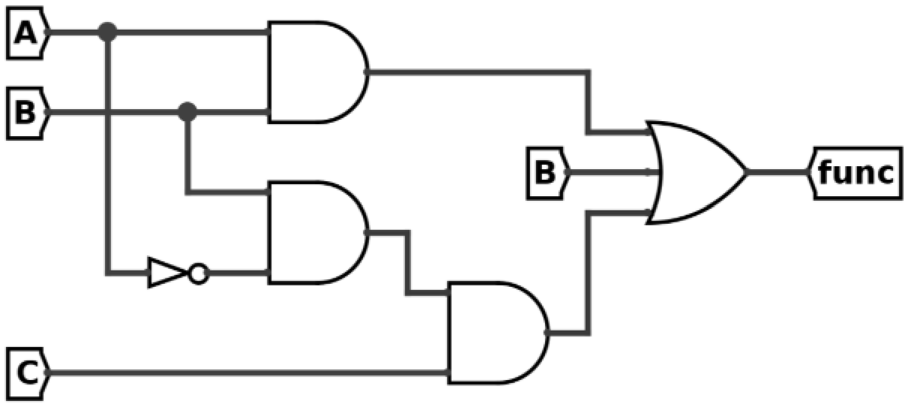


Additionally, $\overline{\mathrm{A}} \cdot \overline{\mathrm{B}}$ is equivalent to a NOR gate and can be redrawn as:



Note that this demonstrates how NOT gates can be formed by feeding the same input to both inputs of a NAND gate. In fact, *all* boolean algebra circuits can be formed only using NAND gates (think about why this may be the case?).

2.3



We can label the outputs of each gate in the circuit. In the first layer, we two AND gates for AB and $\overline{A}B$. The final AND gate takes $\overline{A}B$ as its first input and C as its second input for a combined output of $\overline{A}BC$. Lastly, an OR gate combines the previous gates with the input B for the final function of $F(A, B, C) = AB + \overline{A}BC + B$

Following the procedures in question 1, we can use boolean algebra to simplify the equation:

$$\begin{aligned} AB + \overline{A}BC + B &= \left(A + \overline{A}C + 1\right)B \\ &= B \end{aligned}$$

Thus, we can redraw the circuit as simply:

2.4 Why might it be useful to simplify logic circuits?

Complex digital circuits can be simplified to minimize different objectives such as area or cost. In practice, computers use sophisticated algorithms to optimize circuits for many factors including area, cost, and timing requirements (which will be explored in future week's discussions).

3 Combinational Logic from Truth Tables

For this question, we have a single 3-bit input and a single 4-bit output. We want to design a combinational logic circuit to achieve the desired output given the appropriate combinations of input bits (`Input=001` $\implies$ `Output=0011`, and so on...). Here is the truth table we wish to implement:

Input	**Out**
000	0001
001	0011
010	1111
011-111	xxxx

The `x`'s for the final entry of the table indicate that any output is valid for the case that `Input` is `011`, `100`, `101`, `110`, and `111`

3.1 Write out and simplify boolean expressions for each of the output bits `Out[3], Out[2], Out[1],` and `Out[0]` in terms of the input bits `In[2], In[1], In[0]`.

When deriving expressions for multi-bit values, we find split up the values and find expressions for the individual bits.

Working from right-to-left starting with `Out[0]`, we see that its value is one in all cases which are defined. We can set it to the expression `Out[0] = 1`.

For `Out[1]`, we see that it is `1` whenever `Input=001`, `Input=010`, or for one of the undefined input cases. We can write translate this to an expression as $\text{Out}[1] = \overline{\text{In}[2]}\,\overline{\text{In}[1]}\ \text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]}$. We can also introduce input terms from the undefined cases to help with simplification, namely `Input=101` and `Input=110` to get:

$$\begin{aligned}
\text{Out}[1] &= \overline{\text{In}[2]}\,\overline{\text{In}[1]}\ \text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]} + \text{In}[2]\,\overline{\text{In}[1]}\,\text{In}[0] + \text{In}[2]\,\text{In}[1]\,\overline{\text{In}[0]} \\
&= \overline{\text{In}[2]}\,\overline{\text{In}[1]}\ \text{In}[0] + \text{In}[2]\,\text{In}[1]\,\overline{\text{In}[0]} + \text{In}[2]\,\overline{\text{In}[1]}\,\text{In}[0] + \overline{\text{In}[2]}\,\text{In}[1]\,\overline{\text{In}[0]} \\
&= \overline{\text{In}[1]}\ \text{In}[0] + \text{In}[1]\ \overline{\text{In}[0]}
\end{aligned}$$

To further simplify the expression, we introduce the undefined terms `Input=011` and `Input=111`

$$\begin{aligned}
\text{Out}[1] &= \overline{\text{In}[1]}\ \text{In}[0] + \text{In}[1]\ \overline{\text{In}[0]} + \overline{\text{In}[2]}\ \text{In}[1]\ \text{In}[0] + \text{In}[2]\ \text{In}[1]\ \text{In}[0] \\
&= \overline{\text{In}[1]}\ \text{In}[0] + \text{In}[1]\ \overline{\text{In}[0]} + \text{In}[1]\ \text{In}[0] \\
&= \overline{\text{In}[1]}\ \text{In}[0] + \text{In}[1]\ \overline{\text{In}[0]} + \text{In}[1]\ \text{In}[0] + \text{In}[1]\ \text{In}[0] \\
&= \left(\overline{\text{In}[1]}\ \text{In}[0] + \text{In}[1]\ \text{In}[0]\right) + \left(\text{In}[1]\ \overline{\text{In}[0]} + \text{In}[1]\ \text{In}[0]\right) \\
&= \text{In}[0] + \text{In}[1]
\end{aligned}$$

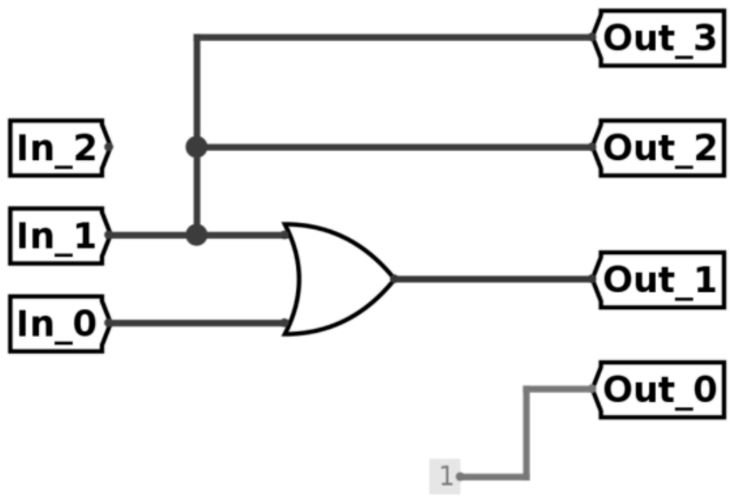
Following a similar process as above, the final two bits simplify to `Out[3] = Out[2] = In[1]`.

While this is a formal way of deriving the answer, there is an alternate, simpler way to approach this problem: think about what combination of input bits can give you the desired output bits.

Notice from the truth table that `Out[1]` is 1 when `In[0]` or `In[1]` is 1, and 0 otherwise. (We ignore the other `In` bits and the undefined cases since they do not matter). Hence, `Out[1] = In[0] + In[1]`.

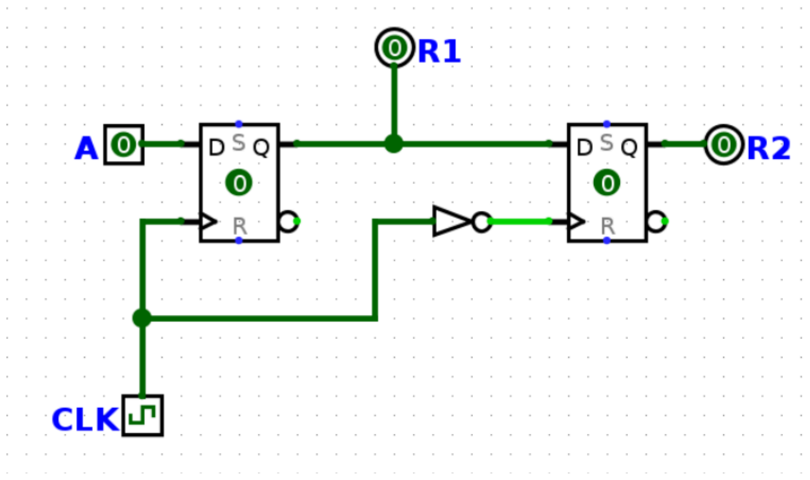
`Out[2]` and `Out[3]` are 1 when `In[1]` is 1, and 0 otherwise. Thus, `Out[3] = Out[2] = In[1]`.

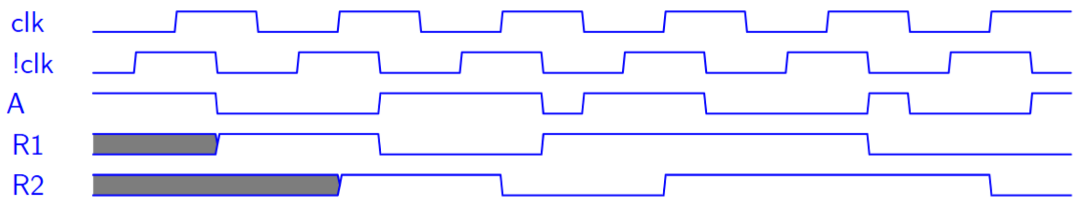
3.2 Draw out the boolean circuit based on your simplified expressions above. You may use constants 0 and 1, and the logic gates AND, OR, NOT.



4 SDS Intro

4.1 Fill out the timing diagram. The clock period (rising edge to rising edge) is 8ps. For every register, clk-to-q delay is 2ps, setup time is 4ps, and hold time is 2ps. NOT gates have a 2ps propagation delay, which is already accounted for in the !clk signal given.

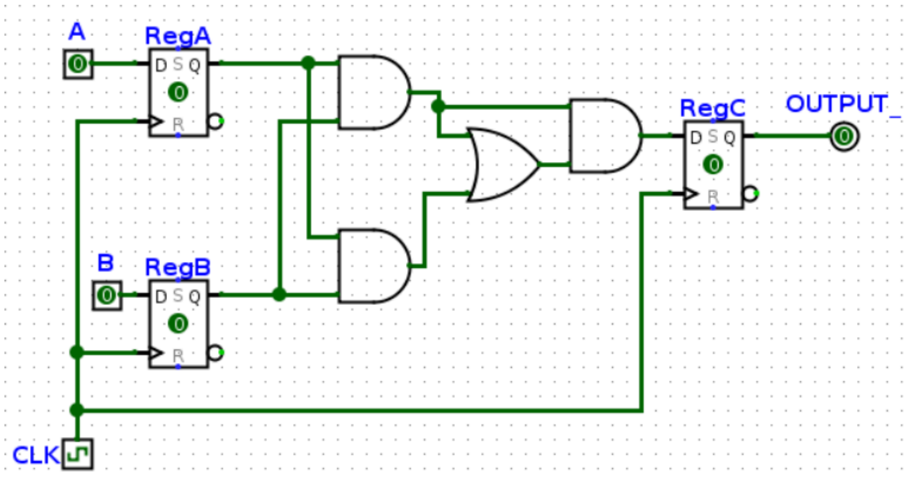




4.2 In the circuit below:

- RegA and RegB have setup, hold, and clk-to-q times of 4ns,
- All logic gates have a delay of 5ns
- RegC has a setup time of 6ns.

What is the maximum allowable hold time for RegC? What is the minimum acceptable clock cycle time for this circuit, and clock frequency does it correspond to?



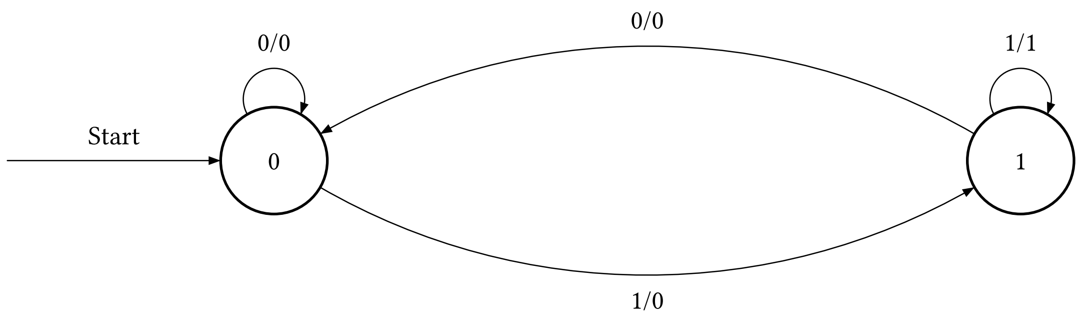
The maximum allowable hold time for RegC is how long it takes for RegC's input to change, so `(clk-to-q of A or B) + shortest CL time = 4 + (5 + 5) = 14 ns`.

The minimum acceptable clock cycle time is `clk-to-q + longest CL time + setup time = 4 + (5 + 5 + 5) + 6 = 25 ns`.

25 ns corresponds to a clock frequency of $\left(\frac{1}{25*10^{-9}}\right)s^{-1} = 40$ MHz

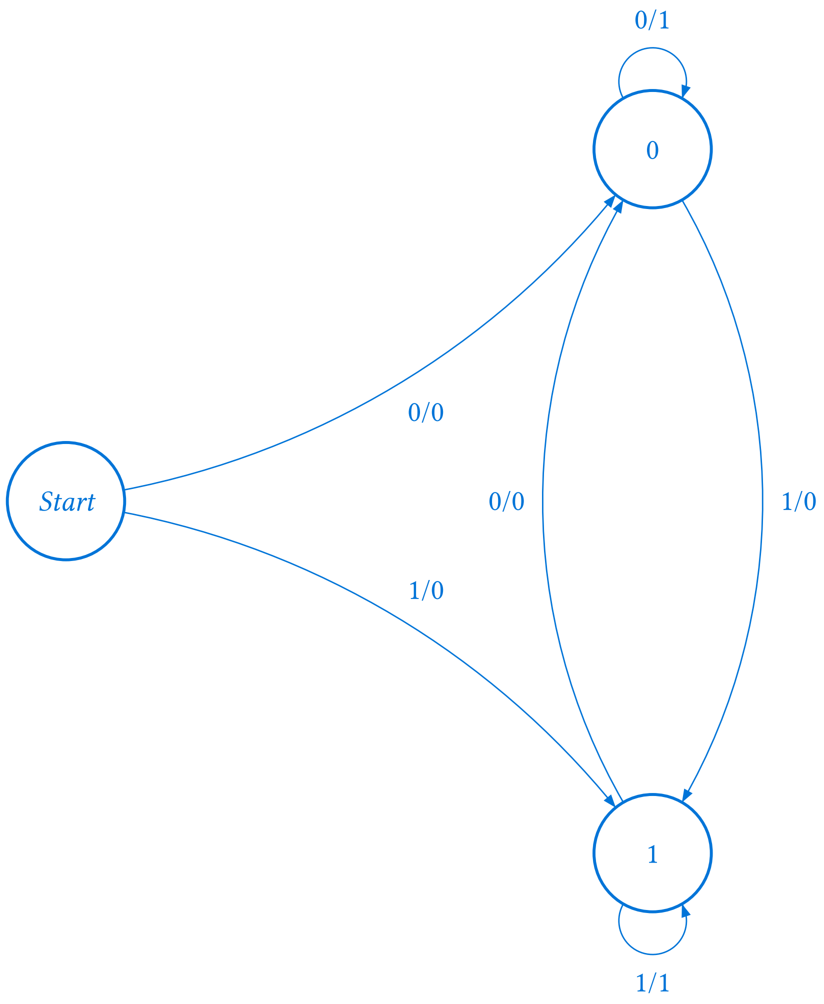
5 FSM

5.1 What pattern in a bitstring does the FSM below detect? What would it output for the input bitstring `011001001110`?

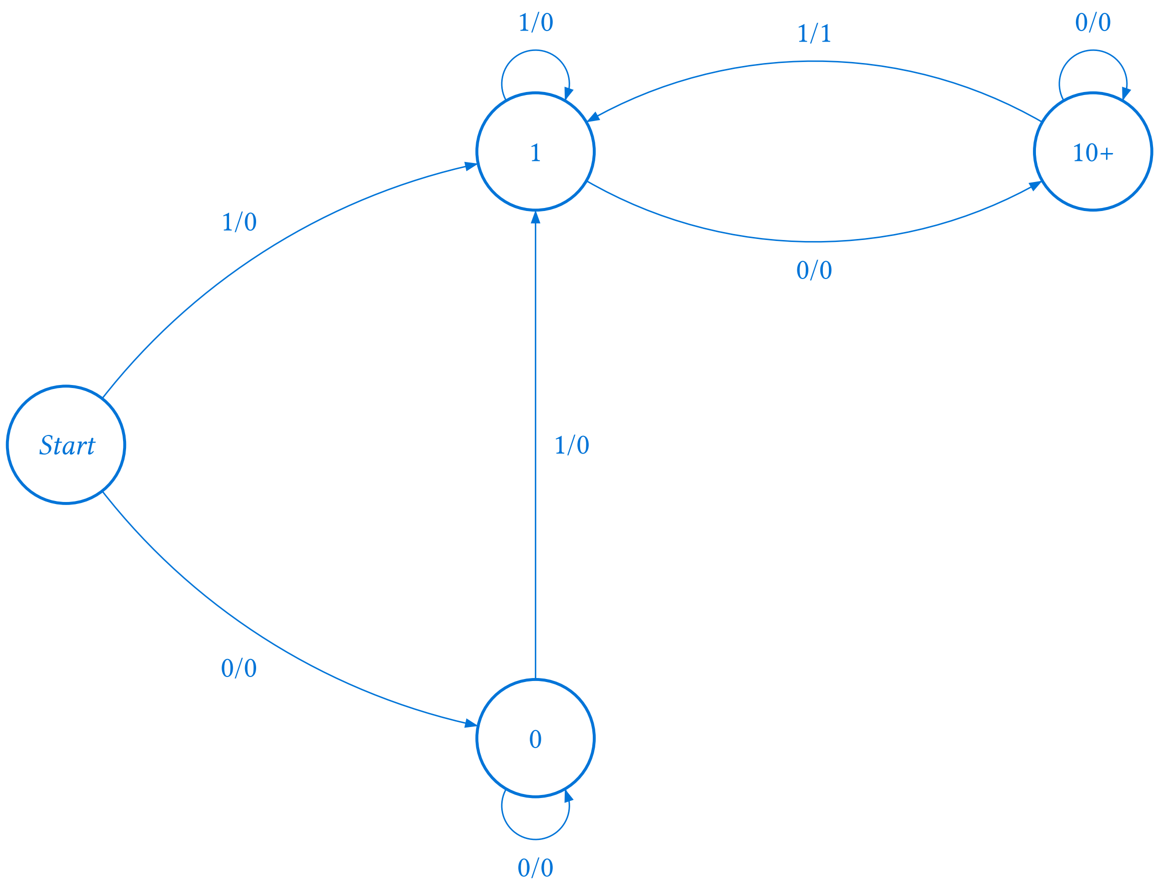


The FSM outputs a `1` if it detects the pattern `11`. The FSM would output `001000000110`.

$\boxed{5.2}$ Fill in the following FSM for outputting a `1` whenever we have two repeating bits as the most recent bits, and a `0` otherwise. You may not need all states.



$\boxed{5.3}$ Draw an FSM that will output a `1` if it recognizes the regex pattern `{10+1}`. That is, if the input forms a pattern of a `1`, followed by one or more `0`s, followed by a `1`.


1/0
1/1
0/0
1
10+
1/0
0/0
Start
1/0
0/0
0
0/0