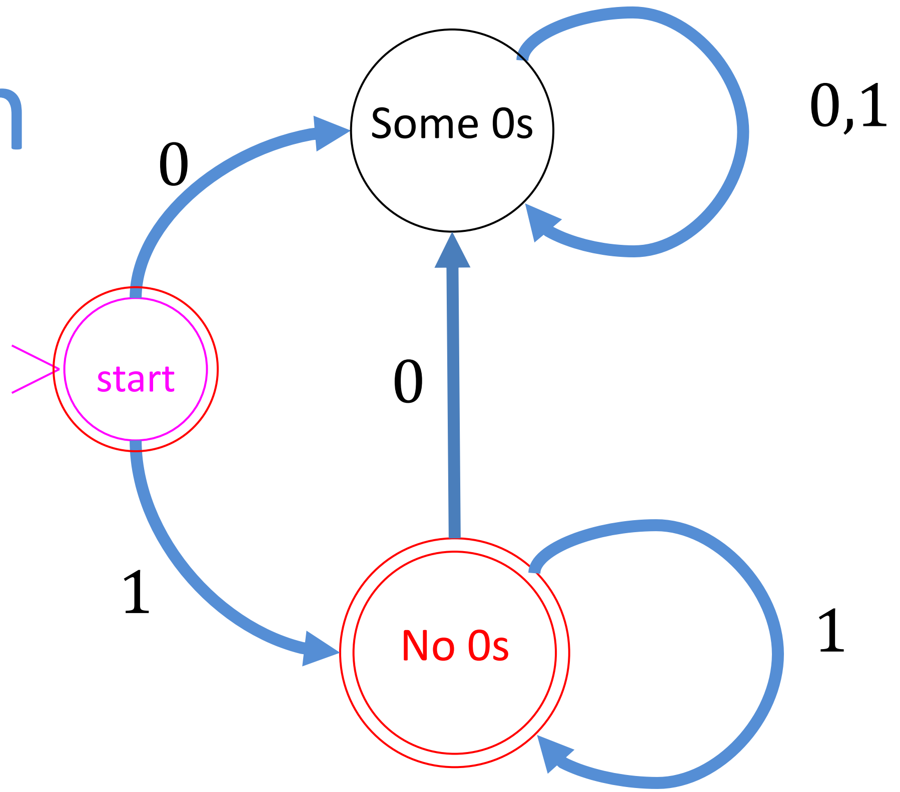
CS3102 Theory of Computation



Warm up:

This automaton computes infinite AND:

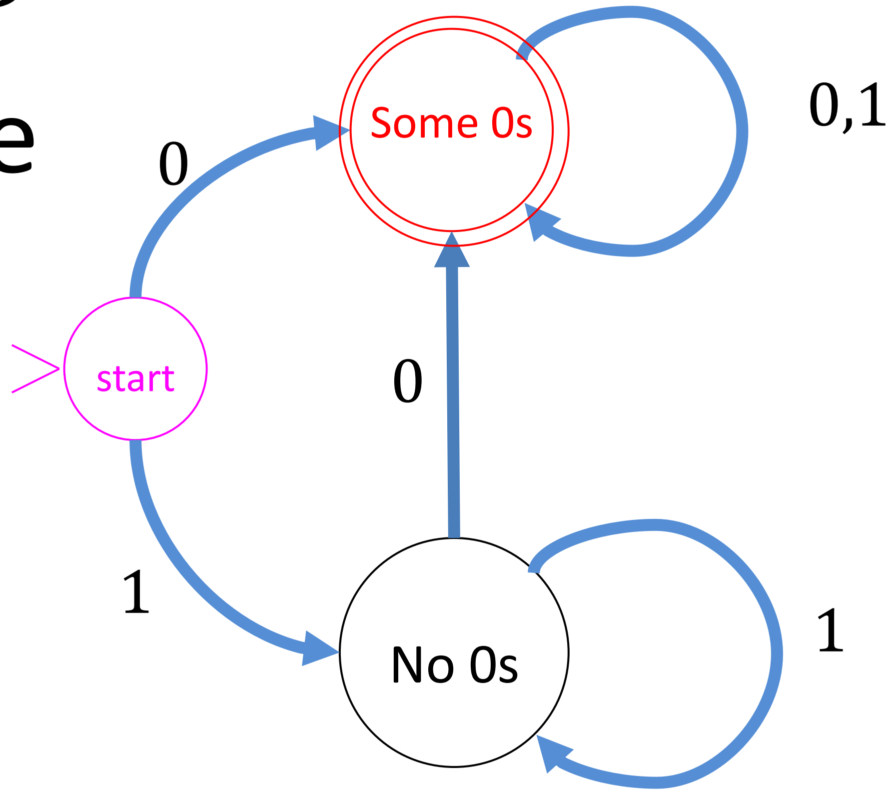
$$AND = \{x \in \{0,1\}^* | x \text{ has no 0s}\}$$

Show how to compute infinite NAND:

$$NAND = \{x \in \{0,1\}^* | x \text{ has a 0}\}$$

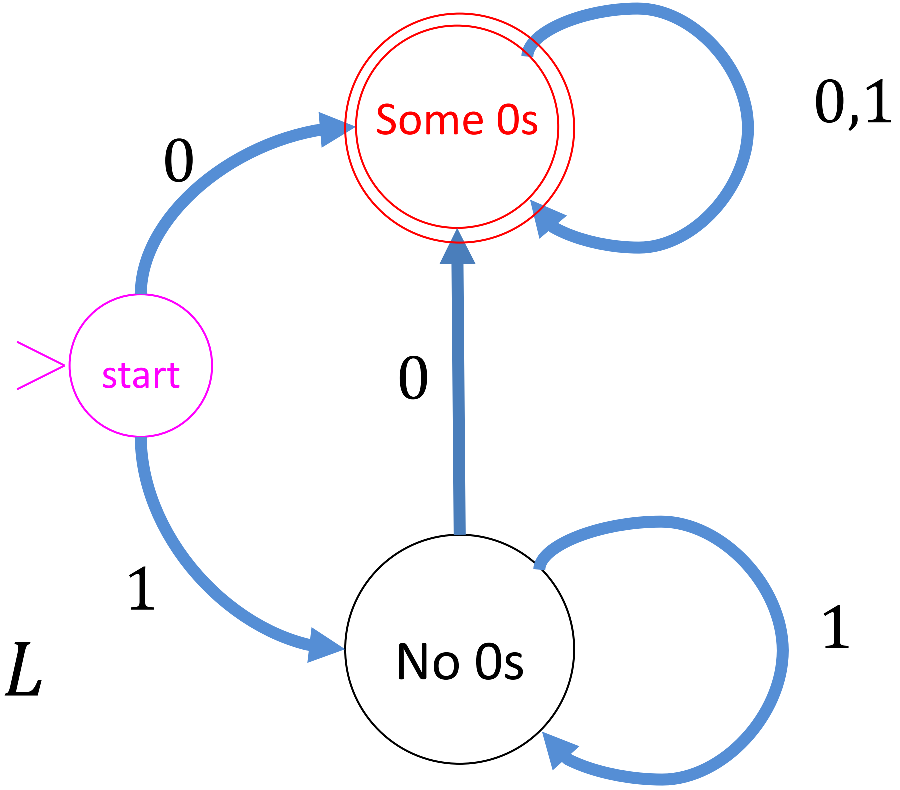
Infinite NAND Automaton

- Observation: $AND^c = NAND$
- NAND should do the opposite of NAND
- Switch final states and non-final states!



AND to NAND

- AND:
  - $Q = \{start, No0s, Some0s\}$
  - $q_0 = start$
  - $F = \{start, No0s\}$
  - δ defined as the arrows
- NAND:
  - Q, q_0, δ don't change
  - $F = Q - F$
- In general, If we can compute a language L with a FSA, we can compute L^c as well



Logistics

- Homework released tomorrow
  - See submission page for deadlines (I’m still processing your quiz 3)
- Quiz will be released Thursday, due Tuesday

Last Time

- Languages and decision problems
  - A different way of thinking about functions
- Introducing Finite State Automata
  - DFA: *Deterministic* finite state automaton
  - Language of a FSA: The set of strings for which that automaton returns 1

FSA are strictly more powerful than NAND circuits

- How can we show this?
  - Show that there is at least one function we can do with FSA but not NAND-CIRC
    - Done! (infinite XOR)
  - Show anything we can do with NAND-CIRC can also be done with FSA
    - How?
    - We need to be able to compute any finite function

Computing any finite function with NAND-CIRC

- Summary:
  - "Manually Precompute" the output for every (finitely-many) possible input
  - When we receive the actual input, do a "lookup"
- Our proof before:
  - Make a variable to represent each possible input, assigning its value to match the correct output
  - Use LOOKUP to return the proper variable for the given input

Straightline Code for *f*

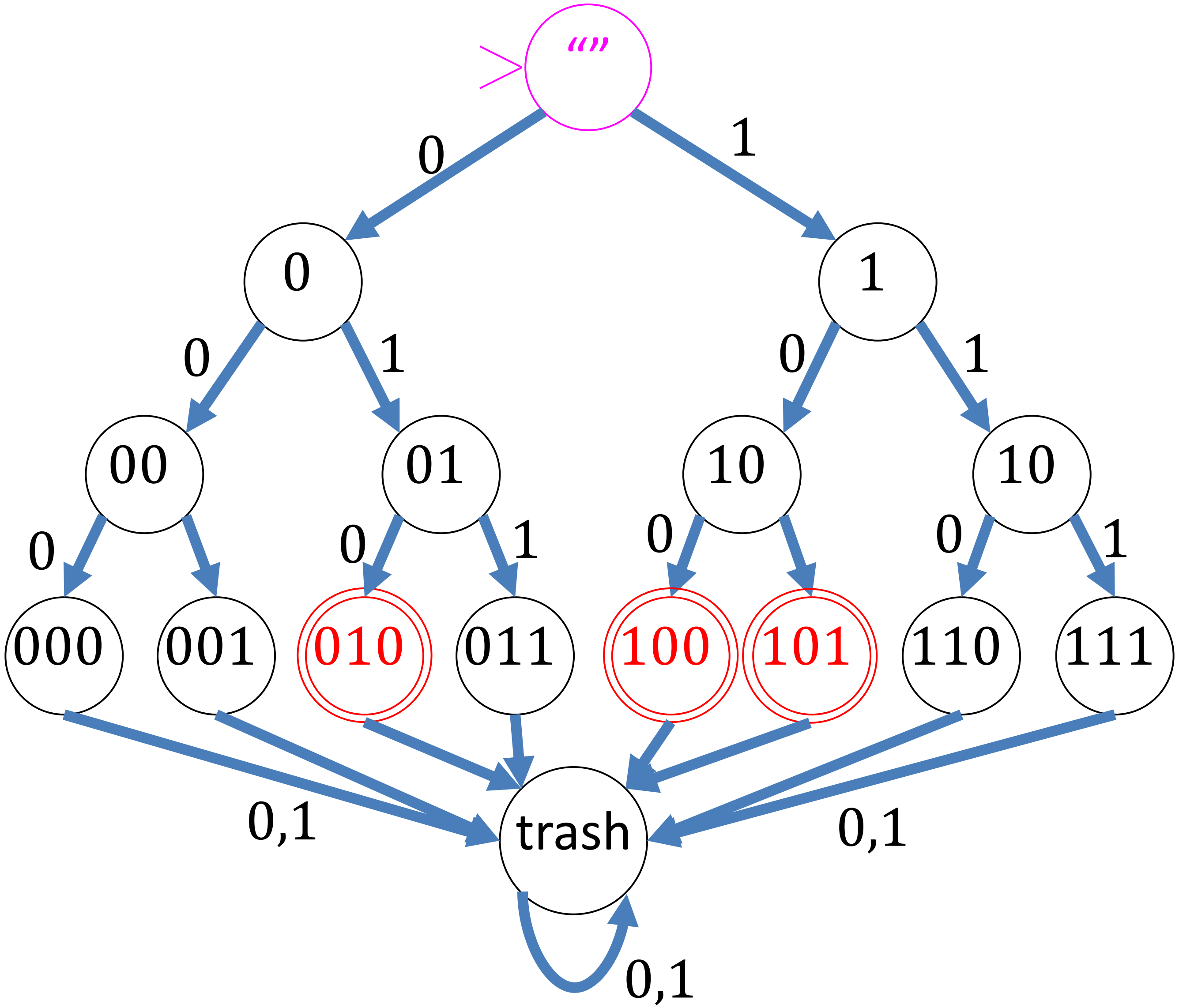
Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

```
def F(x0,x1,x2):
    F000=0
    F001=0
    F010=1
    F011=0
    F100=1
    F101=1
    F110=0
    F111=1
    return LOOKUP3(F000,F001,F010,F011,F100,F101,F110,F111,x0,x1,x2)
```

Computing finite functions with FSA

- Summary:
  - "Manually Precompute" the output for every (finitely-many) possible input
  - When we receive the actual input, do a "lookup"
- Same idea, but with Automata:
  - Make a state for every possible input, determining whether or not it is final depending on the correct output
  - Do a "binary tree traversal" with the given input to navigate to its correct output

FSA for f



Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

Regular Expressions

Name	Decision Problem	Function	Language
Regex	Does this string match this pattern?	$f(b) = \begin{cases} 0 & \text{the string matches} \\ 1 & \text{the string doesn't} \end{cases}$	$\{b \in \Sigma^* \mid b \text{ matches the pattern}\}$

- A way of describing a language
- Give a "pattern" of the strings, every string matching that pattern is in the language
- Examples:
  - $(a|b)c$ matches : ac and bc
  - $(a|b)^*c$ matches : $c, ac, bc, aac, abc, bac, bbc, \ldots$

“Pieces” of a Regex

- Empty String:
  - Matches just the string of length 0
  - Notation: ε or “”
- Literal Character
  - Matches a specific string of length 1
  - Example: the regex a will match just the string a
- Alternation/Union
  - Matches strings that match at least one of the two parts
  - Example: the regex $a|b$ will match a and b
- Concatenation
  - Matches strings that can be dividing into 2 parts to match the things concatenated
  - Example: the regex $(a|b)c$ will match the strings ac and bc
- Kleene Star
  - Matches strings that are 0 or more copies of the thing starred
  - Example: $(a|b)c^*$ will match a, b, or either followed by any number of c’s

Note: The compents here are the minimal necessary. In practice, regexes have other components as well, those are just “syntactic sugar”.

Regex for UVA computing IDs

- A UVA computing id is formatted as:
  - 2-3 letters
  - A digit
  - 1-3 letters

AND as a Regex

- $AND = \{x \in \{0,1\}^* | x \text{ has no 0s}\}$

NAND as a Regex

- $NAND = \{x \in \{0,1\}^* | x \text{ has a } 0\}$

XOR as a Regex

- $XOR = \{x \in \{0,1\}^* | x \text{ has an odd number of 1s}\}$

FSA = Regex

- Finite state Automata and Regular Expressions are equivalent models of computing
- Any language I can represent as a FSA I can also represent as a Regex (and vice versa)
- How would I show this?

Showing FSA $\leq$ Regex

- Show how to convert any FSA into a Regex for the same language
- We're going to skip this:
  - It's tedious, and people virtually never go this direction in practice, but you can do it (see textbook theorem 9.12)

Showing Regex $\leq$ FSA

- Show how to convert any regex into a FSA for the same language
- Idea: show how to build each "piece" of a regex using FSA

“Pieces” of a Regex

- Empty String:
  - Matches just the string of length 0
  - Notation: ε or “”
- Literal Character
  - Matches a specific string of length 1
  - Example: the regex a will match just the string a
- Alternation/Union
  - Matches strings that match at least one of the two parts
  - Example: the regex $a|b$ will match a and b
- Concatenation
  - Matches strings that can be dividing into 2 parts to match the things concatenated
  - Example: the regex $(a|b)c$ will match the strings ac and bc
- Kleene Star
  - Matches strings that are 0 or more copies of the thing starred
  - Example: $(a|b)c^*$ will match a, b, or either followed by any number of c’s

FSA for the empty string

FSA for a literal character

FSA for Alternation/Union

- Tricky…
- What does it need to do?