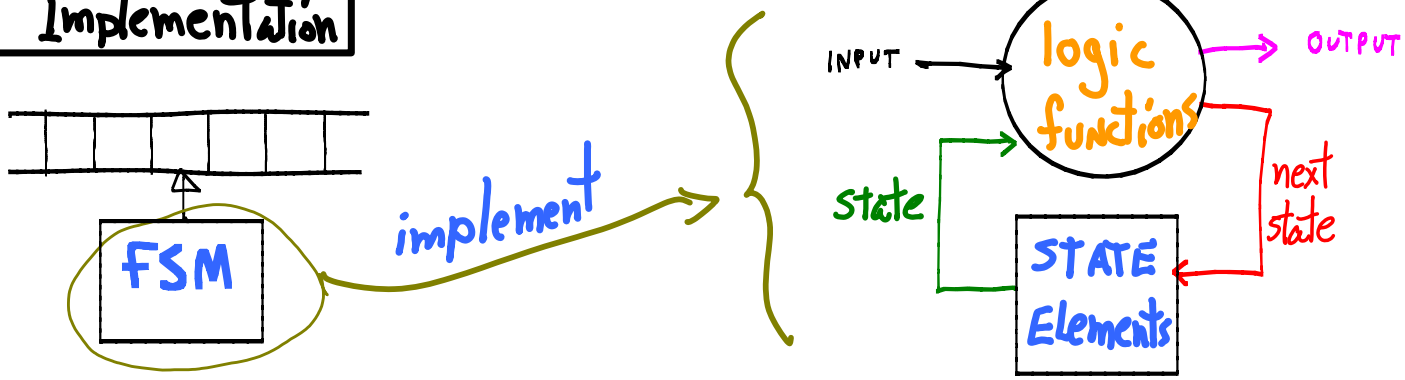
TM Implementation



To **build any TM, WE NEED**:

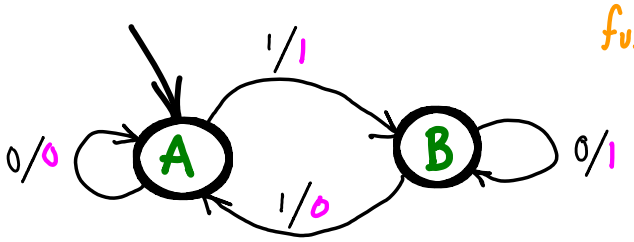
--- **(1.) FSM**:
state
logic functions (**output** and **next-state**)

--- **(2.) Tape**: **methods to R/W symbols**, we'll use registers (RAM).

--- **(3.) Symbol set** = a set of fixed length bit strings, e.g.,
S = {0,1} (**2 symbols**)
S = {00, 01, 10, 11} (**4 symbols**)
S = {000, 001, 010, 011, 100, 101, 110, 111} (**8 symbols**)

Build FSM

even-odd parity



{symbols} = {0, 1}

{states} = {A, B} encode → {0, 1}

functions
OUT: {states} X {symbols} → {symbols}
next_state: {states} X {symbols} → {states}

OUT		encode		OUT
(A, 0)	===> 0	→	(0, 0)	===> 0
(A, 1)	===> 1		(0, 1)	===> 1
(B, 0)	===> 1		(1, 0)	===> 1
(B, 1)	===> 0		(1, 1)	===> 0

next_state		encode		next_state
(A, 0)	===> A	→	(0, 0)	===> 0
(A, 1)	===> B		(0, 1)	===> 1
(B, 0)	===> B		(1, 0)	===> 1
(B, 1)	===> A		(1, 1)	===> 0

current STATE
symbol read
next state

Boolean functions

Two things to build to implement a FSM:

1. Boolean functions
2. State elements

Our Computer/Simulator is a FSM.
Our simulated machines are FSMs, too.

Turing Complete/Universal (can **simulate any TM**) we need:
--- a **Language** to **describe any TM**,
--- a **Simulator** that **understands that language**.

We need to describe arbitrary TMs so that our computer/simulator can execute/simulate them.

We can build the machines physically.
We need a language to describe them as well.

LANGUAGE

Our description language must be able to describe:

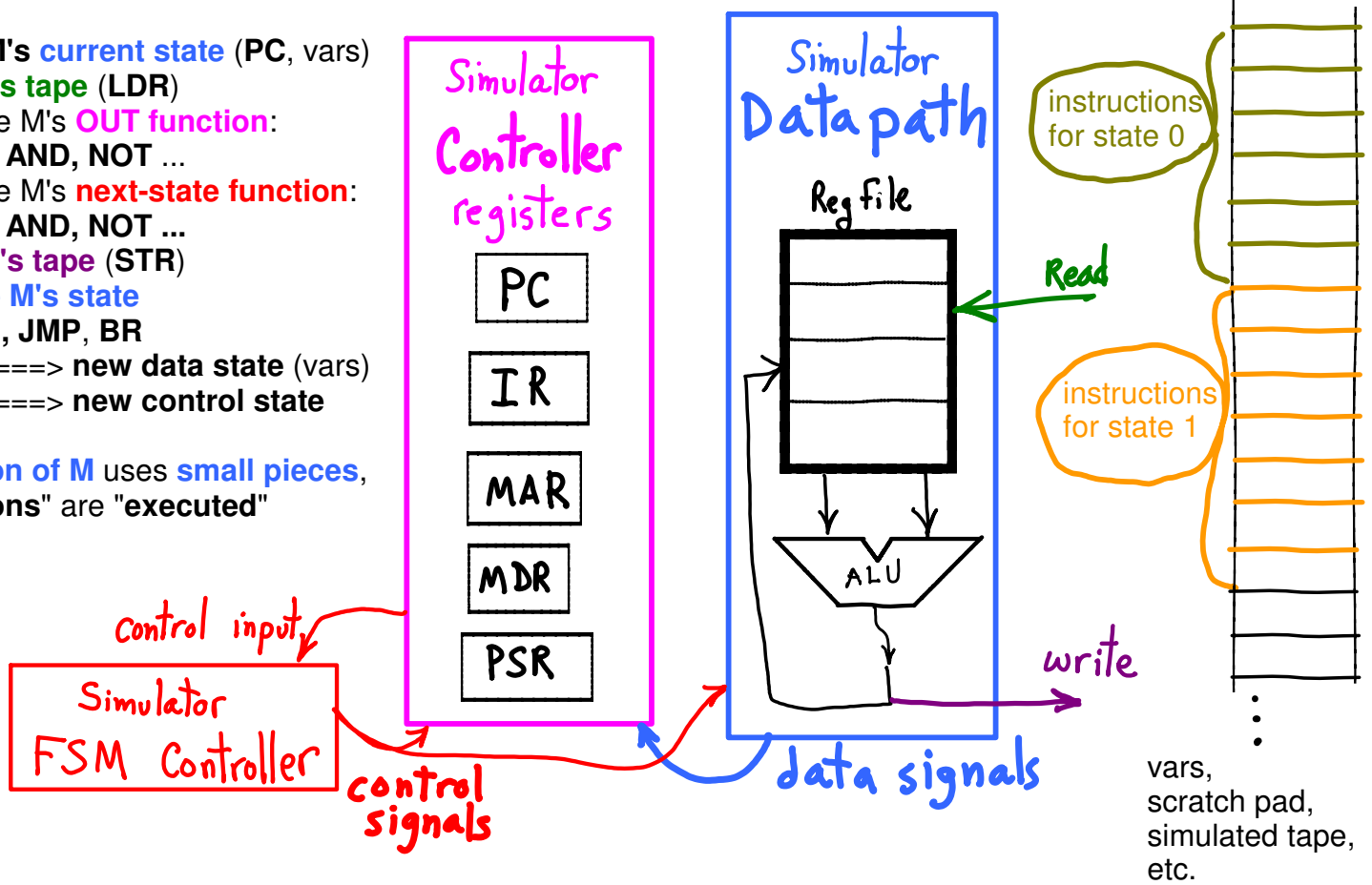
--- Arbitrary **set of states**, including regs (= vars)
--- Arbitrary **set of symbols**
--- Arbitrary **branching** **(via binary trees)**
--- **RW to tape**
--- Arbitrary **functions** (**next-state**, **output**)

UTM/Simulator/computer

A physical computer/simulator has to be able to do only a small number of steps to execute/simulate a program/description of some TM M.

--- **know M's current state** (**PC**, vars)
--- **read M's tape** (**LDR**)
--- compute M's **OUT function**:
ADD, AND, NOT ...
--- compute M's **next-state function**:
ADD, AND, NOT ...
--- **write M's tape** (**STR**)
--- **change M's state**
PC++, JMP, BR
STR ===> new data state (vars)
STR ===> new control state

Description of M uses **small pieces**, "**instructions**" are "**executed**"



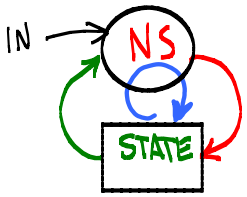
FSM Controller uses **registers** (e.g., PC) to **remember**:
--- **M's state** (control + data states)
--- **symbols read** (RegFile)
--- **symbols to write** (RegFile)
--- **step of simulation** **(UTM's controller's state)**
--- **partial steps** of **function evaluations** (next-state, output)
data registers, PSR, on tape, ...

STATE ELEMENTS

pos. edge-triggered FF

Problem:

Katy bar the door



Feedback loop:
state change,
NS change,
state change,
NS change, ...

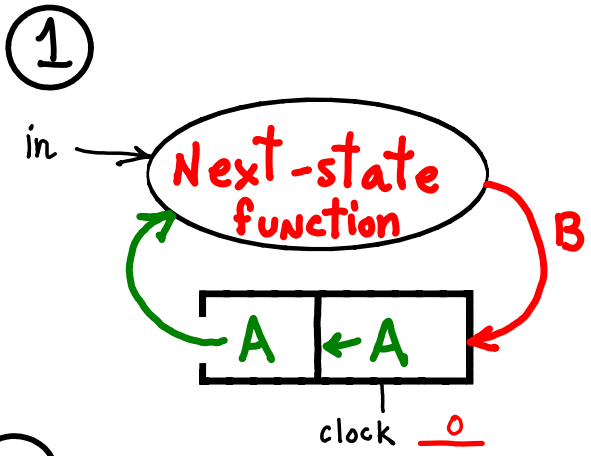
State changes w/o control.

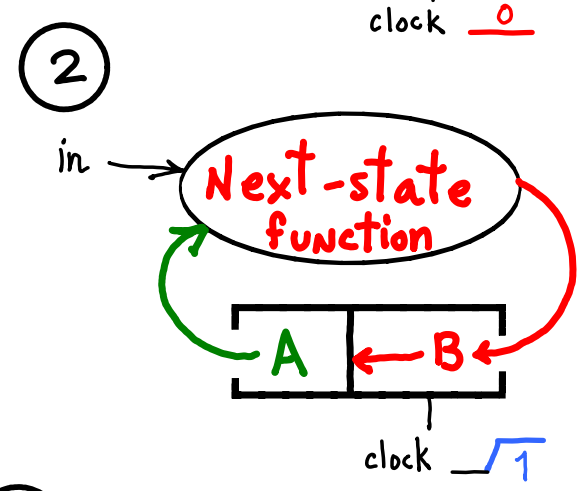
We want coordination w/ input data.

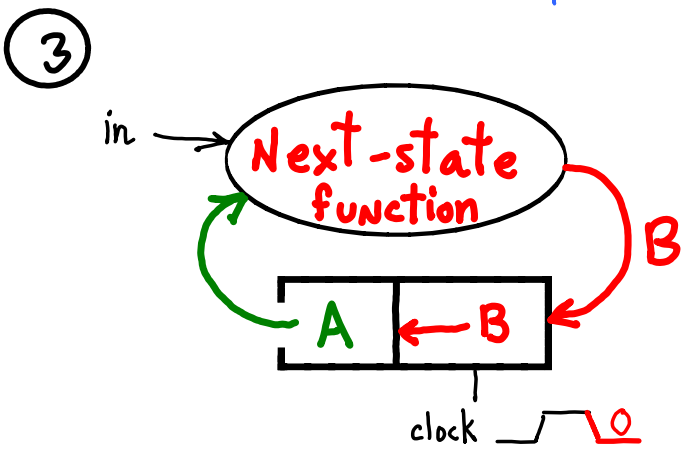
When the input is ready, Then allow a state change.

Input has to be stable for a minimum time.

Current state has to remain stable to get correct next state.







Solution

Never allow a complete cycle path.

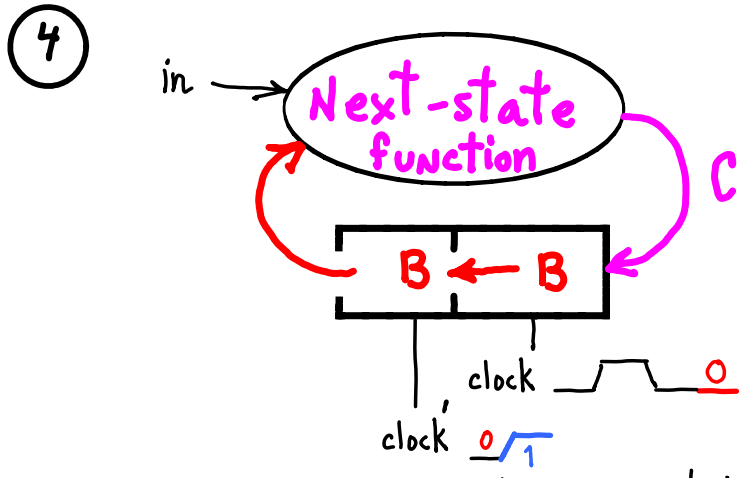
Break path at the state element.

1. Current state = **A**

2. Allow NS signal **B** in (read/sample state element input).

3. Then, close the barn door.
B is captured, cannot be changed by changes in NS function output.

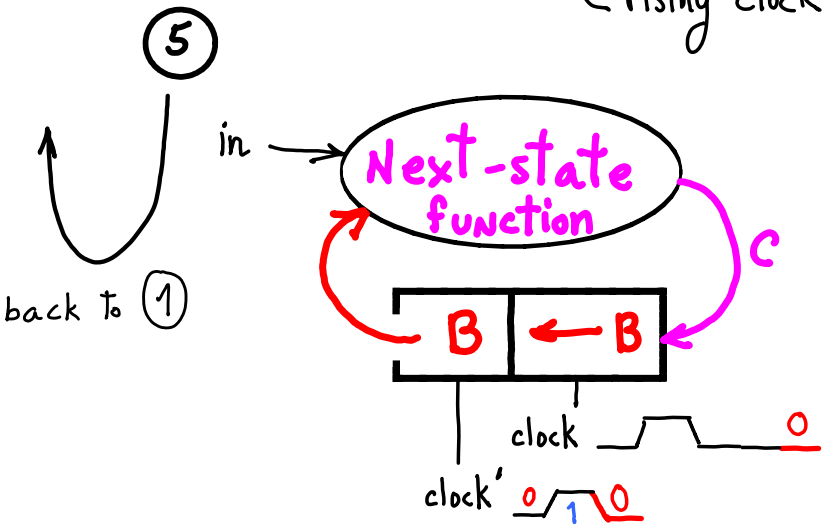
input "sampled"
"latched"



Let the NS out:
open the output side barn door,
== change current state to B

NS function immediately responds with new next-state C.

↖ rising clock = state change

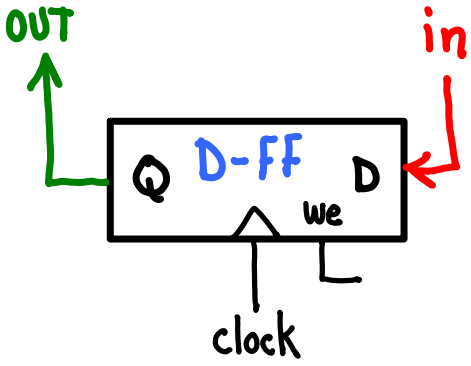


Close the output side's input door.

New state B is latched.

Path is never a complete cycle.

This is what we need for implementing STATE:
a D-FF register.



Note:
We add a "write-enable" because sometimes we need to choose whether or not the register will change its value.
So far, for FSMs, our D-FFs are always write-enabled, we==1.

Describing Functions

Two Ways to Specify a Function $f(x)$

---

(1)

Describe how to evaluate $f(x)$:
E.g.

$$f(x) = 2x$$

(given any value x)

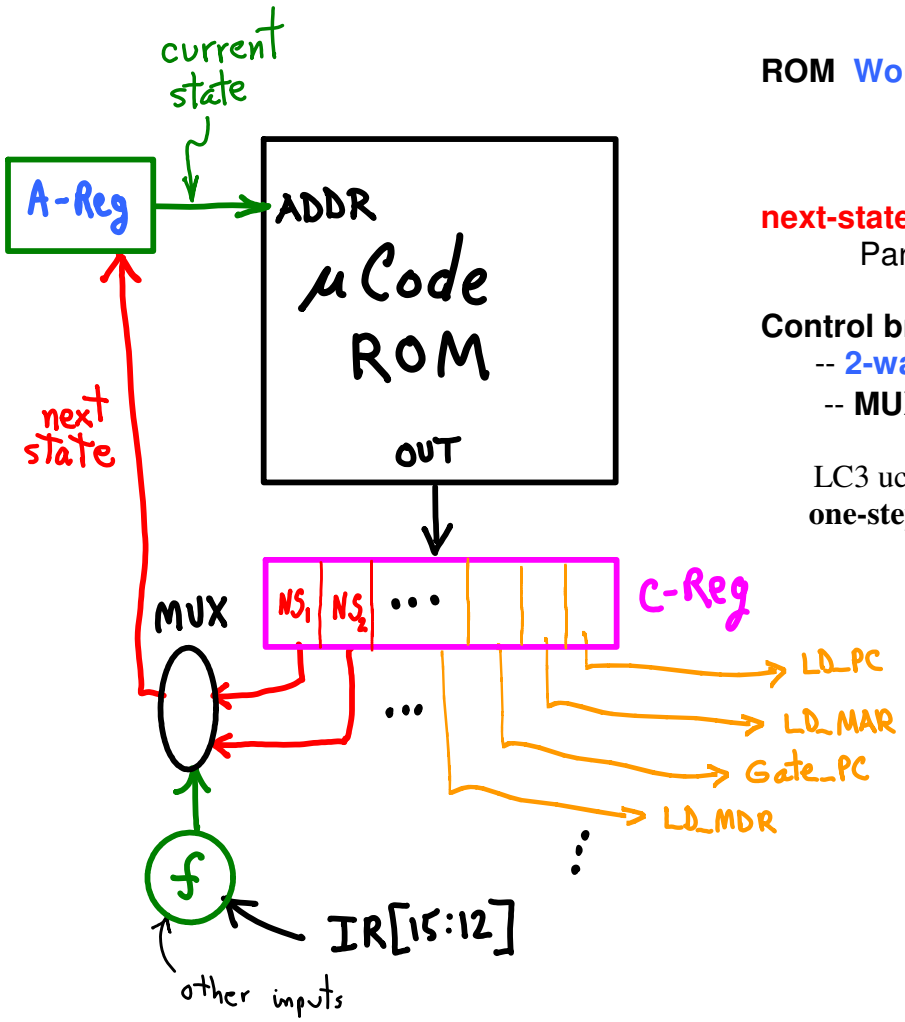
(2)

Show a table of values of $f(x)$:
E.g.

x	$f(x)$
0	1
1	0

(for every value of x)

(B) microcoded controller next-state & output functions



A-Reg == **controller's current state**
addresses **uCode ROM**
gives memory **word at output**

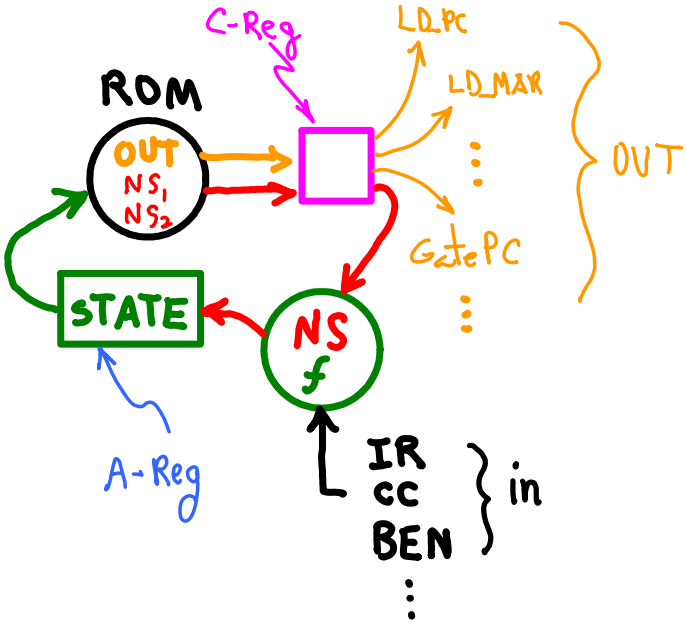
ROM Word == **datapath control bits**
C-Reg has **current control word**
controls datapath

next-state fields of C-Reg
Part of **Controller's next-state** function

Control branching (the rest of **next-state** function)
-- **2-way**, **NS1** or **NS2**
-- **MUX.select** = f (**STATE**, **IR**, **...)**

LC3 ucode branching also includes
one-step, 16-way branching (DECODE).

As a general FSM,
it looks like this,



Advantages of **ucode controller:**

-- easier to **change**
-- easier to **figure out**
-- easier to **expand**
install bigger ROM.

Advantages of "**random logic**" **controller**:

-- **faster**
-- **smaller** (?)
-- **distributed throughout machine**

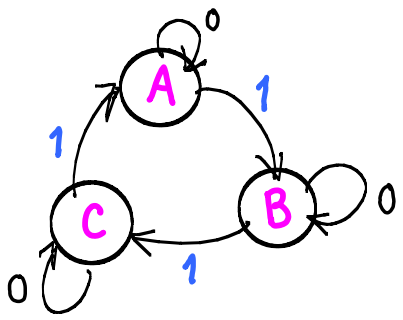
Caveat: The C-Reg is just to make the picture clearer, it doesn't actually exist in LC3. Instead, the ROM's outputs go directly to control inputs.

Suppose we have,
--- **1-bit state elements**
--- **1-bit function elements**

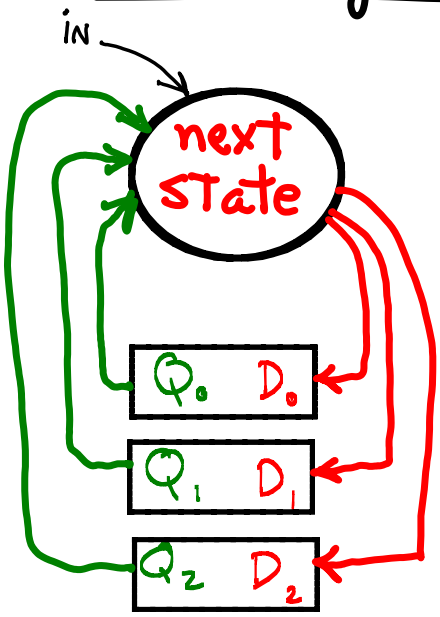
HOW do we **put them together to implement a FSM M**?

We need to **encode the states of M** in bits.
We get **boolean functions** for **next-state function** (and **output** function).

e.g., mod-3 machine



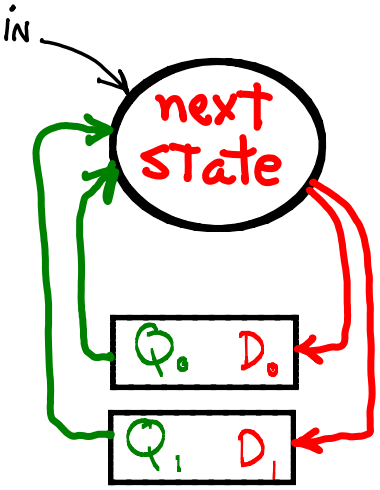
1-hot encoding 3 state elements



encoding

STATE	Q_2	Q_1	Q_0
A	0	0	1
B	0	1	0
C	1	0	0

binary encoding 2 state elements



encoding

STATE	Q_1	Q_0
A	0	0
B	0	1
C	1	0

The next-state function as a table:

x = input, current state (IN, Q2, Q1, Q0) ; $f(x)$ = next state (D2, D1, D0)

IN	Q2	Q1	Q0		D2	D1	D0	
0	0	0	1	(A)	0	0	1	(A)
1	0	0	1	(A)	0	1	0	(B)
0	0	1	0	(B)	0	1	0	(B)
1	0	1	0	(B)	1	0	0	(C)
0	1	0	0	(C)	1	0	0	(C)
1	1	0	0	(C)	0	0	1	(A)
*	*	*	*		X	X	X	

* rows that cannot be reached.
X is for **don't care**, either 0 or 1.

next state

IN	Q1	Q0		D1	D0	
0	0	0	(A)	0	0	(A)
1	0	0	(A)	0	1	(B)
0	0	1	(B)	0	1	(B)
1	0	1	(B)	1	0	(C)
0	1	0	(C)	1	0	(C)
1	1	0	(C)	0	0	(A)
*	*	*		X	X	

* rows cannot be reached.
X is for **don't care**: either 0 or 1.

IF we have a **universal language** (able to describe **any TM**)

All we need to know is **How To Build**:

--- **1-bit state elements**?

--- **1-bit functions**?