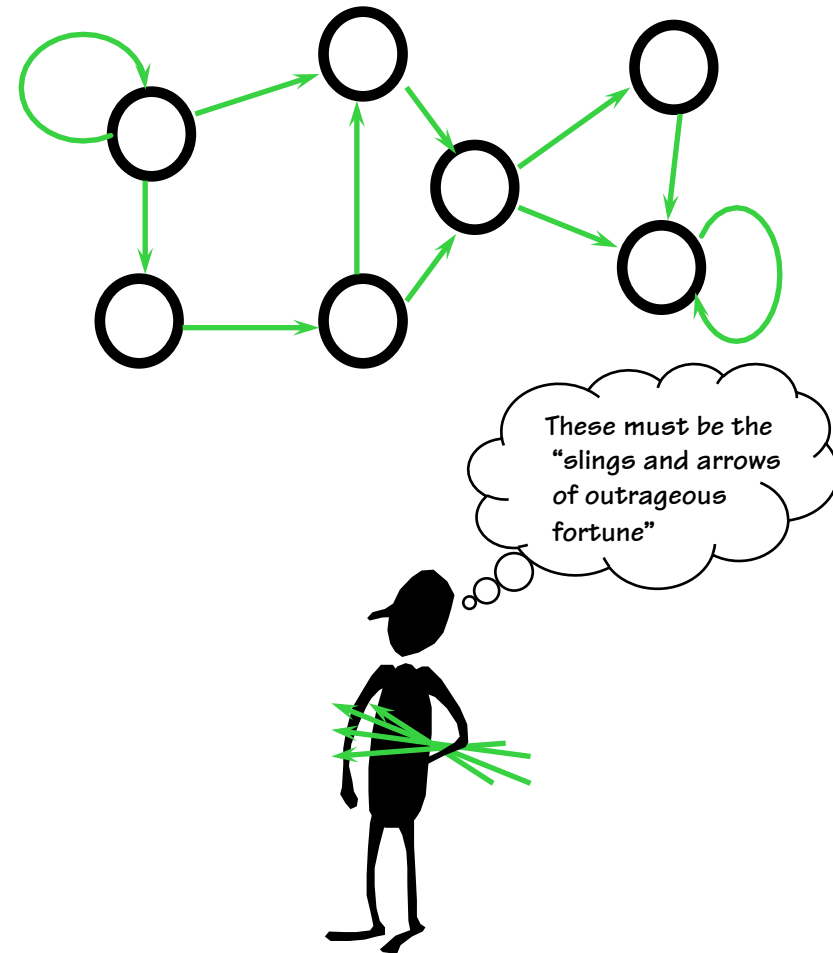
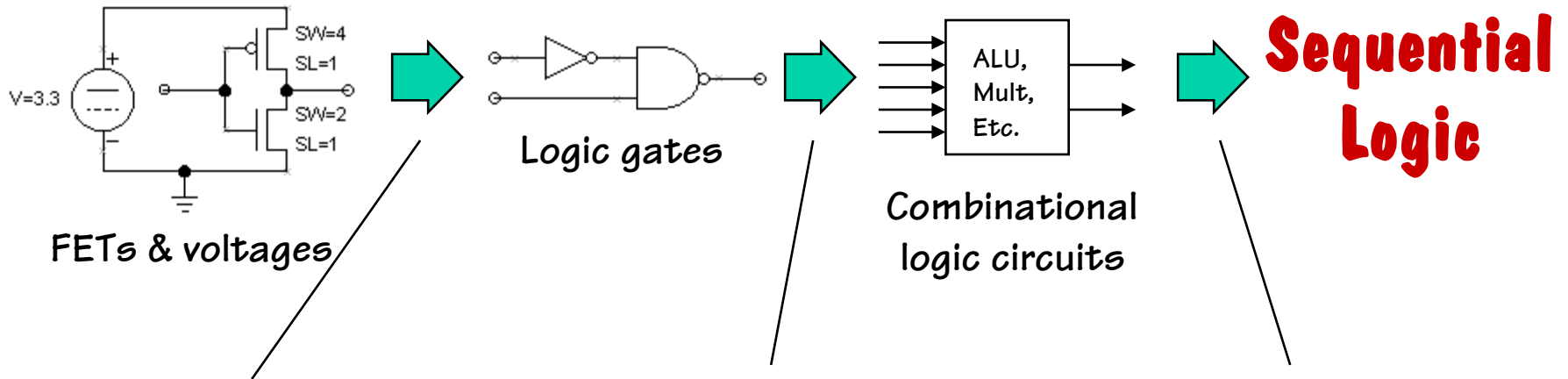


Synchronous Logic

- 1) Sequential Logic
- 2) Synchronous Design
- 3) Synchronous Timing Analysis
- 4) Single Clock Design
- 5) Finite State Machines
- 6) Turing Machines
- 7) What it means to be “Computable”



Road Traveled So Far...



Combinational contract:

- ◆ Voltage-based “bits”
- ◆ 1-bit per wire
- ◆ Generate quality outputs, tolerate inferior inputs
- ◆ Combinational contract
- ◆ Complete in/out/timing spec

Acyclic connections

Composable blocks

Design:

- ◆ truth tables
- ◆ sum-of-products
- ◆ muxes
- ◆ ROMs

Storage & state

Dynamic discipline

Finite-state machines

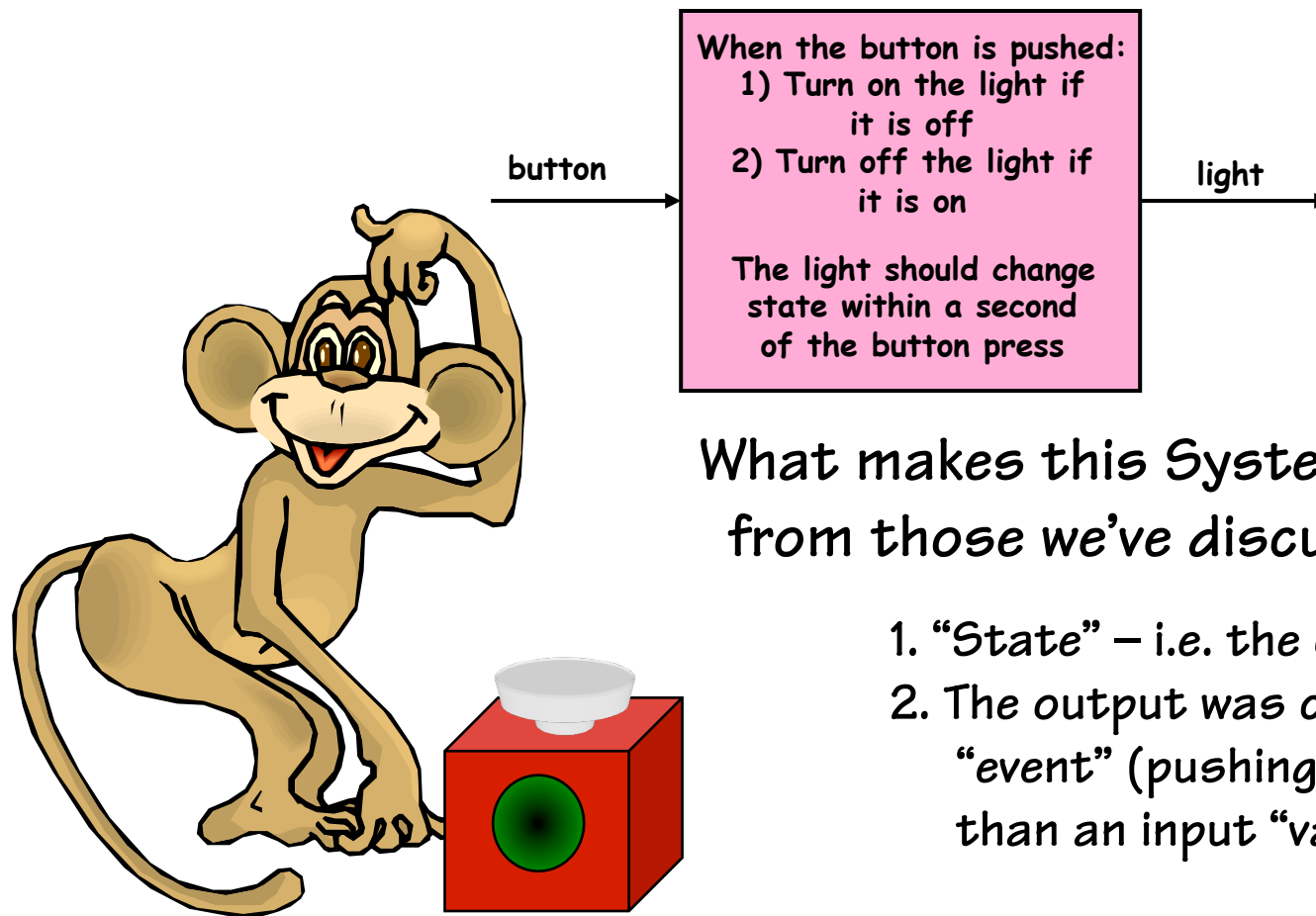
Throughput & latency

Pipelining

Our motto: *Sweat the details once, and then put a box around it!*

Something We Can't Build (Yet)

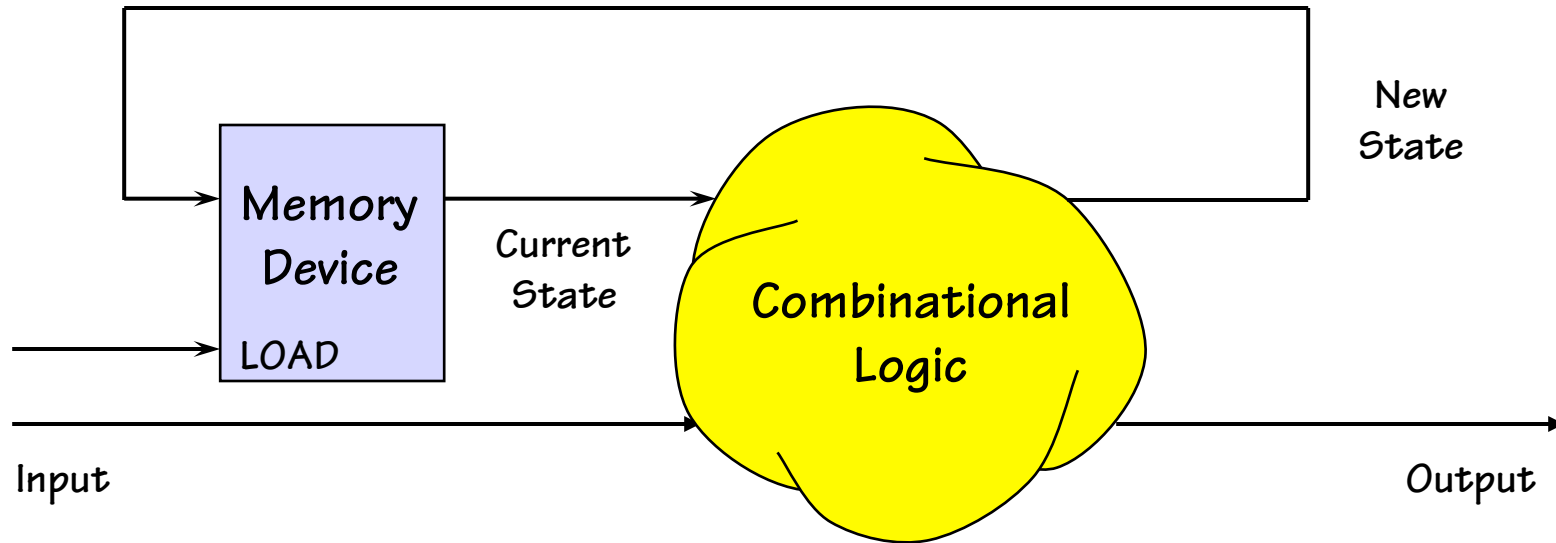
What if you were given the following system design specification?



What makes this System so different from those we've discussed before?

1. "State" – i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

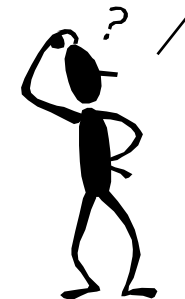
"Sequential" = Stateful



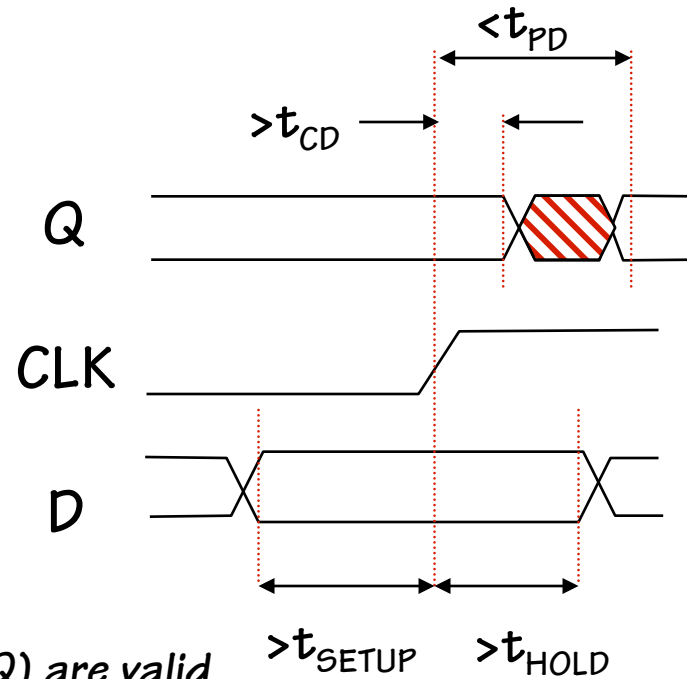
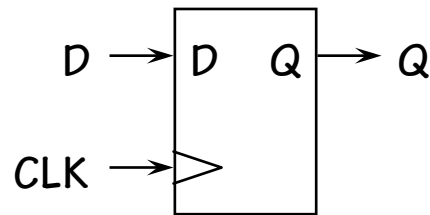
Plan: Build a Sequential Circuit with stored digital STATE –

- MEMORY stores CURRENT state
- Combinational Logic computes
 - the NEXT state (Based on inputs & current state)
 - the OUTPUTs (Based on inputs and/or current state)
- State changes on LOAD control input

Didn't we develop some memory devices last time?



Review of Flip Flop Timing



t_{PD} : maximum propagation delay, CLK \rightarrow Q
 How LONG after clock rises until outputs (Q) are valid

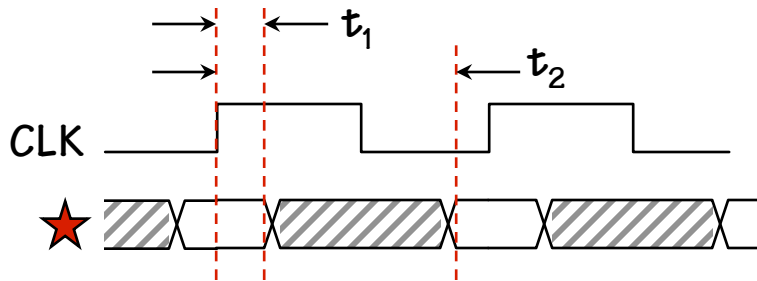
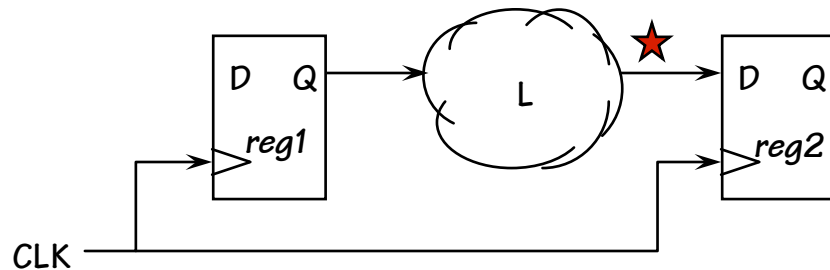
t_{CD} : minimum contamination delay, CLK \rightarrow Q
 How SOON after clock rises until outputs (Q) go invalid

t_{SETUP} : setup time
 How LONG data (D) input must be stable before clock's rising edge

t_{HOLD} : hold time
 How LONG data (D) inputs must be held after clock's rising edge

We haven't explicitly mentioned this timing attribute, but it must have existed even for combinational logic. We can always safely assume it is 0 (i.e. the outputs become invalid immediately)

"Synchronous" Timing Analysis



$$t_1 = t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$$

$$t_2 = t_{PD,reg1} + t_{PD,L} \leq t_{CLK} - t_{SETUP,reg2}$$

$$\text{Minimum Clock Period: } t_{CLK} \geq t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2}$$

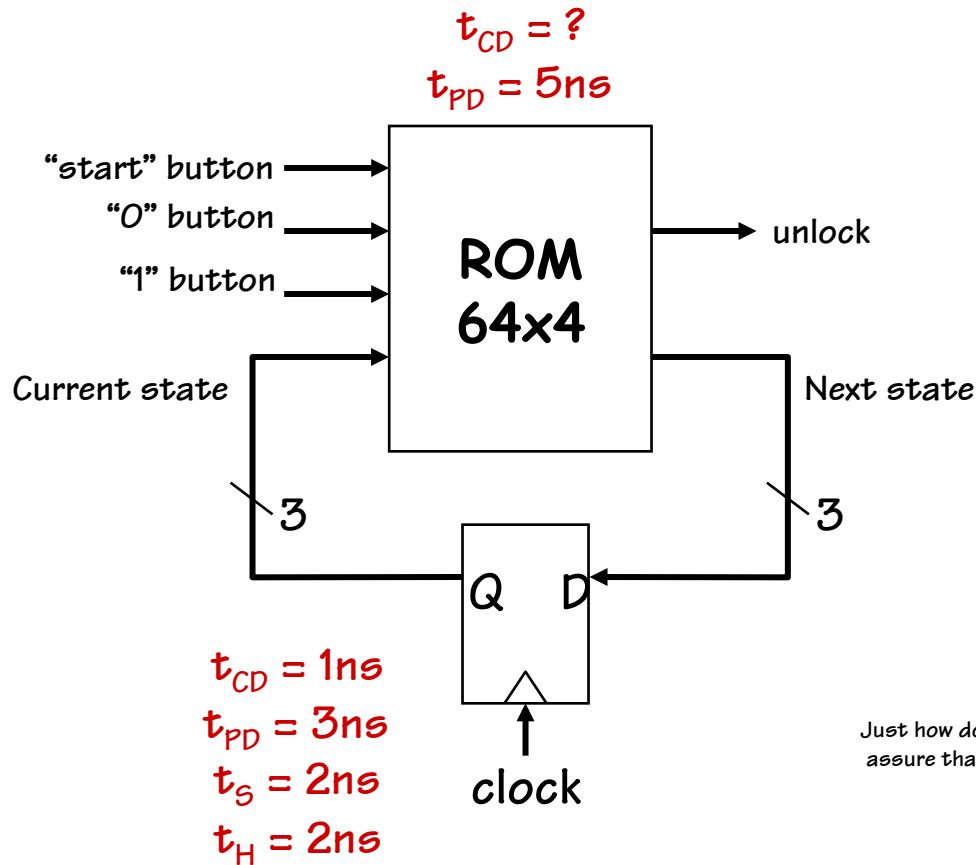
Questions for register-based designs:

- ◆ How much time for useful work (i.e. for combinational logic delay)?
- ◆ Does it help to guarantee a minimum t_{CD} ? How 'bout designing registers so that

$$t_{CD,reg} \geq t_{HOLD,reg}?$$

- ◆ What happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

Example: Flip Flop Timing



Questions:

1. t_{CD} for the ROM?

$$t_{CD,REG} + t_{CD,ROM} > t_{H,REG}$$

$$1ns + t_{CD,ROM} > 2ns$$

$$t_{CD,ROM} > 1ns$$

2. Min. clock period?

$$t_{CLK} > t_{PD,REG} + t_{PD,ROM} + t_{S,REG}$$

$$t_{CLK} > 3ns + 5ns + 2ns$$

$$t_{CLK} > 10ns$$

3. Constraints on inputs?

"start", "0", and "1" must be valid before the clock and held

$$t_{PD,ROM} + t_{S,REG} = 5 + 2 = 7ns$$

after it.

$$t_{H,REG} - t_{CD,ROM} = 2 - 1 = 1ns$$

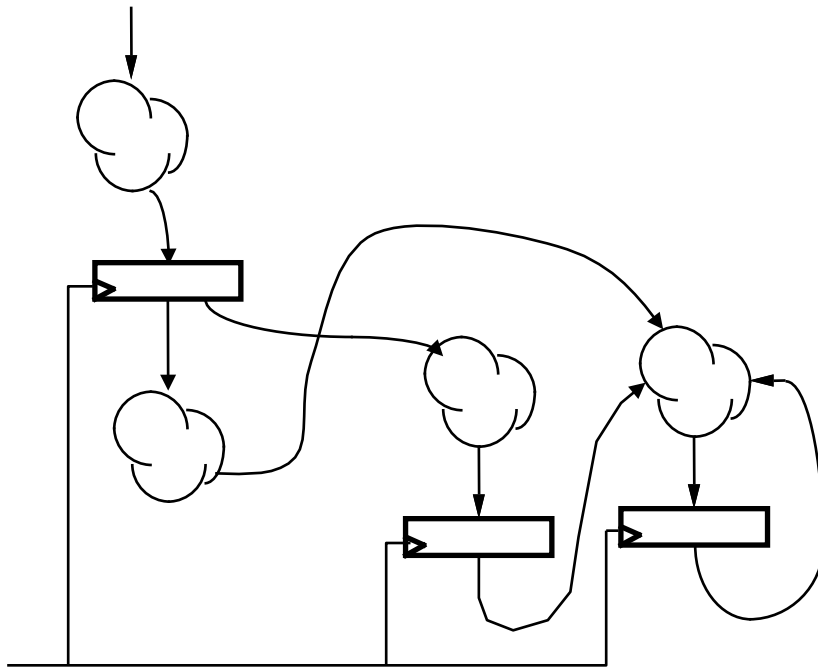
Just how do I assure that?



Single Synchronous Clock Design

Sequential \neq Synchronous

However, Synchronous = A recipe for robust sequential circuits:



- **No combinational cycles**
(other than those already built into the registers)
- **Only cares about values of combinational circuits just before rising edge of clock**
- **Clock period greater than every combinational delay**
- **Changes state after all logic transitions have stopped!**

Designing Sequential Logic

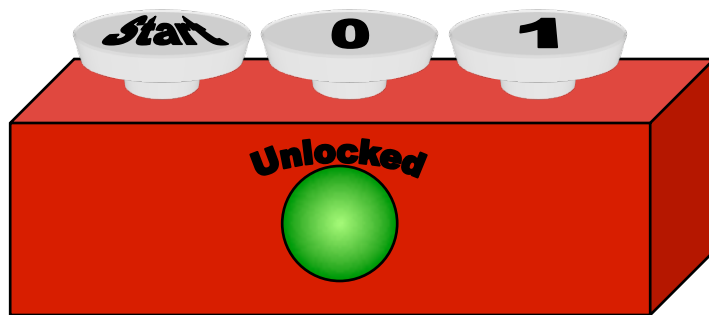
Sequential logic is used when the solution to some design problem involves a *sequence of steps*:

How to open digital combination lock w/ 3 buttons (“start”, “0” and “1”):

- Step 1: press “start” button
- Step 2: press “0” button
- Step 3: press “1” button
- Step 4: press “1” button
- Step 5: press “0” button



Information remembered between steps is called **state**. Might be just what step we’re on, or might include results from earlier steps we’ll need to complete a later step.



Implementing a "State Machine"

This flavor of "truth-table" is called a "state transition table"

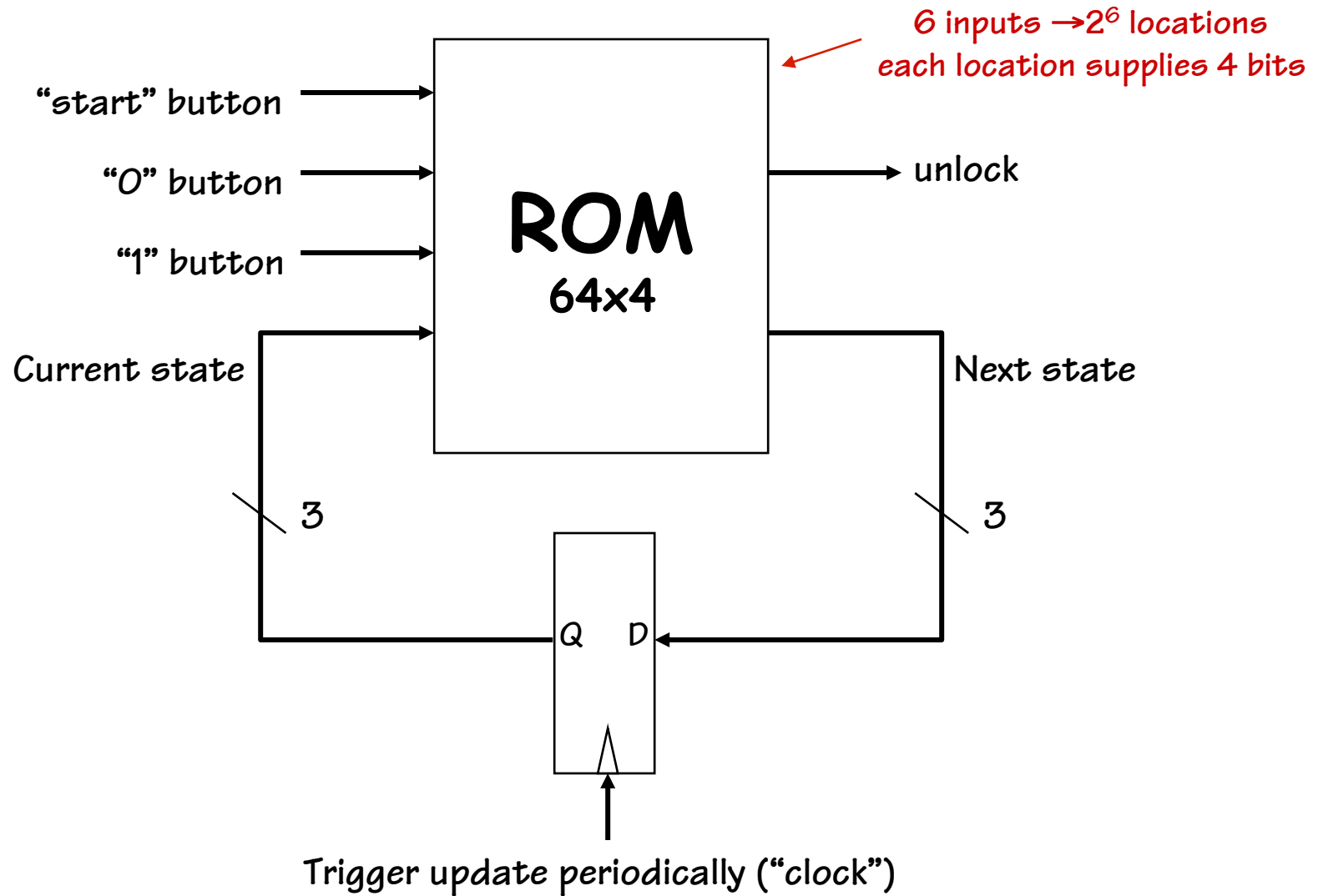
Current state	"start"	"1"	"0"	Next state	unlock
---	1	---	---	start 000	0
start 000	0	0	1	digit1 001	0
start 000	0	1	0	error 101	0
start 000	0	0	0	start 000	0
digit1 001	0	1	0	digit2 010	0
digit1 001	0	0	1	error 101	0
digit1 001	0	0	0	digit1 001	0
digit2 010	0	1	0	digit3 011	0
...					
digit3 011	0	0	1	unlock 100	0
...					
unlock 100	0	1	0	error 101	1
unlock 100	0	0	1	error 101	1
unlock 100	0	0	0	unlock 100	1
error 101	0	---	---	error 101	0

This is starting to look like a PROGRAM

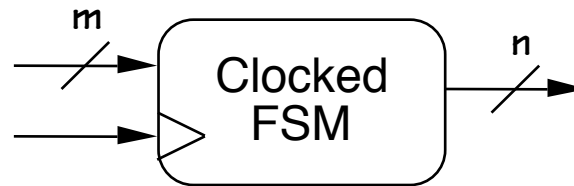


6 different states → encode using 3 bits

Now Do It With Hardware!



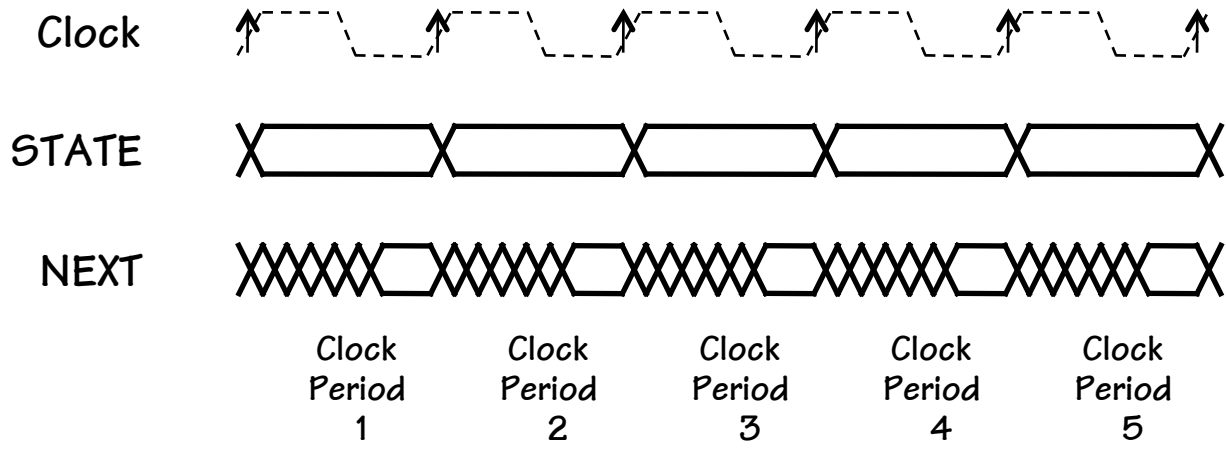
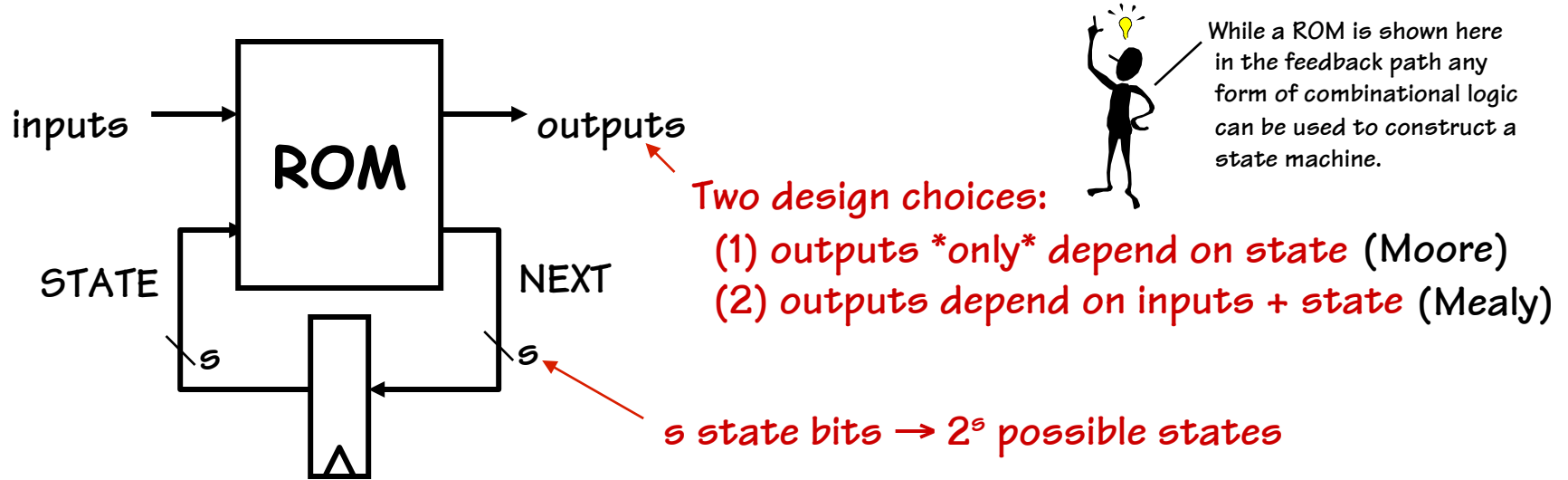
Abstraction du jour: Finite State Machines



A FINITE STATE MACHINE has

- k STATES $S_1 \dots S_k$ (one is "initial" state)
- m INPUTS $I_1 \dots I_m$
- n OUTPUTS $O_1 \dots O_n$
- Transition Rules $S'(S,i)$
for each state S and input i
- Output Rules $Out(S)$ for each state S

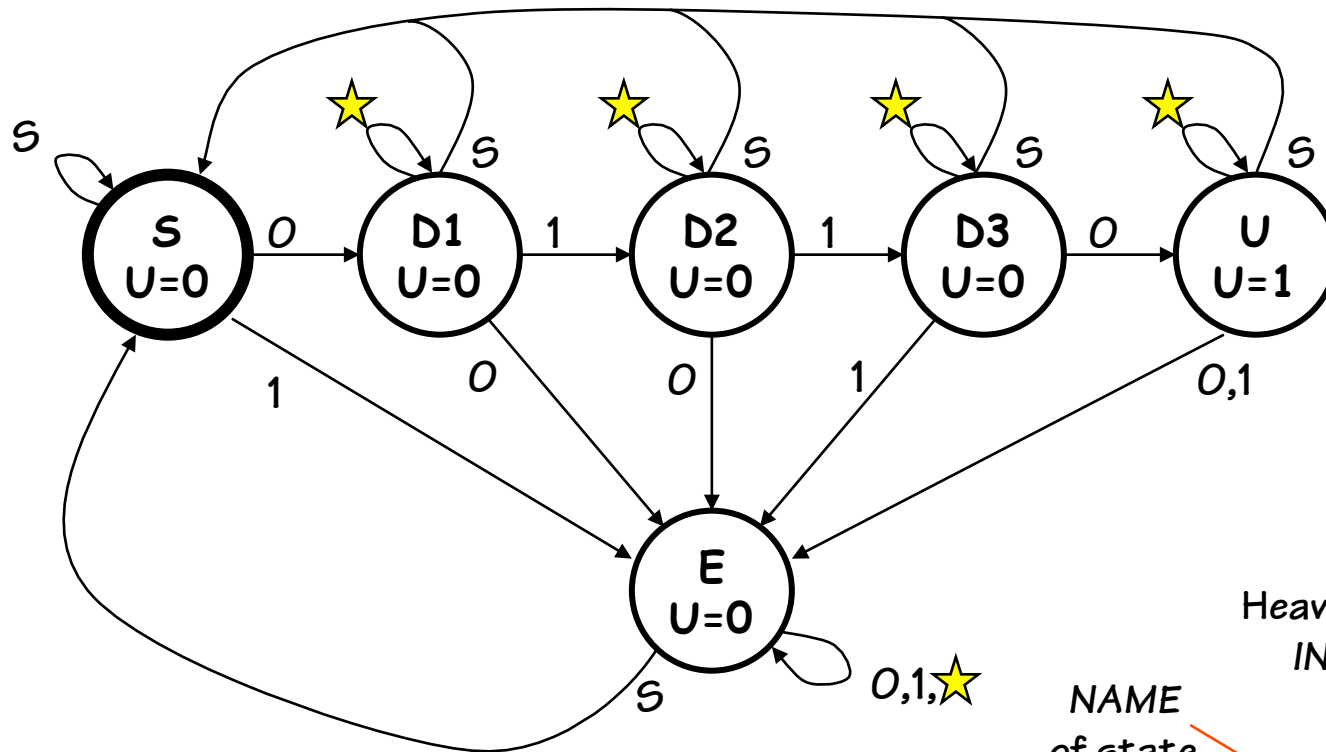
Discrete State, Time



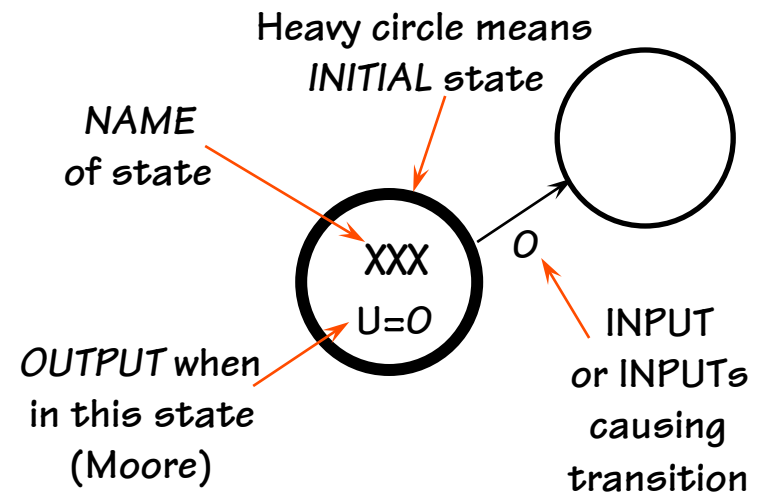
State Transition Diagrams

A state transition diagram is an abstract “graph” representation of a “state transition table”, where each state is represented as a node and each transition is represented as an arc.

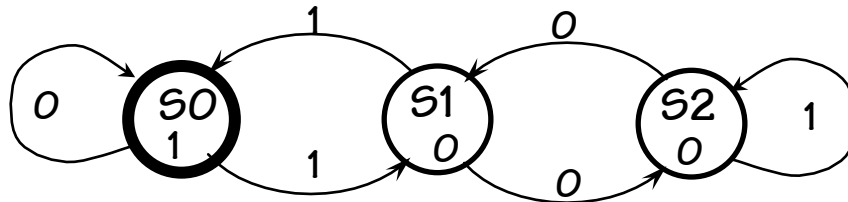
It represents the machine's behavior not its implementation!



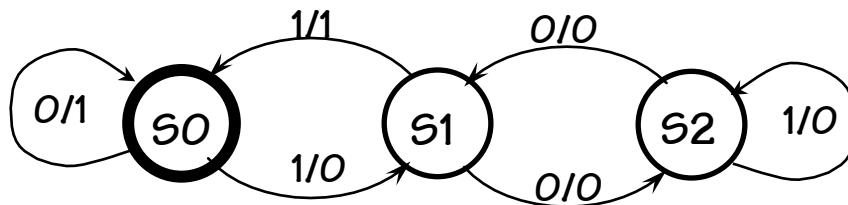
★ = no buttons pressed



Valid State Diagrams



MOORE Machine:
Outputs on States



MEALY Machine:
Outputs on Transitions

Arcs leaving a state must be:

(1) **mutually exclusive**

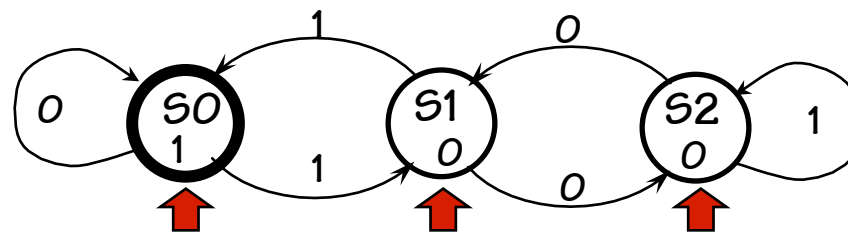
can only have one choice for any given input value

(2) **collectively exhaustive**

every state must specify what happens for each possible input combination. "Nothing happens" means arc back to itself.

Let's Play State Machine

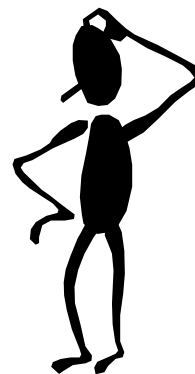
Let's emulate the behavior specified by the state machine shown below when processing the following string from LSB to MSB.



$$39_{10} = 0100111_2$$

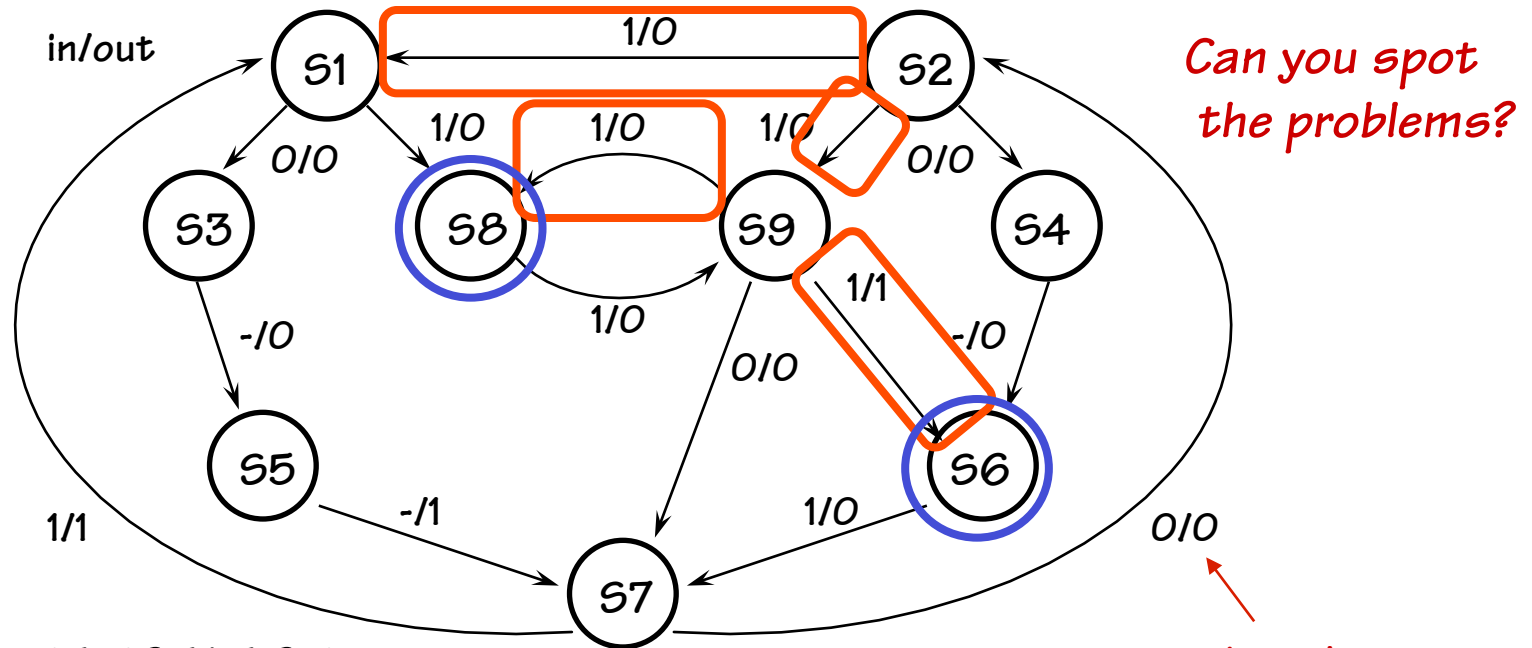
← input order

	State	Input	Next	Output
T=0	S0	1	S1	0
T=1	S1	1	S0	1
T=2	S0	1	S1	0
T=3	S1	0	S2	0
T=4	S2	0	S1	0
T=5	S1	1	S0	1
T=6	S0	0	S0	1



It looks to me like this machine outputs a 1 whenever the bit sequence that it has seen thus far is a multiple of 3. (Wow, and FSM can divide by 3!)

Busted Stuff



Can you spot the problems?

input/output (Mealy)

CONVENIENT NOTATION:

When a transition is made on the next input regardless of its value the arc can be labeled with an X or -

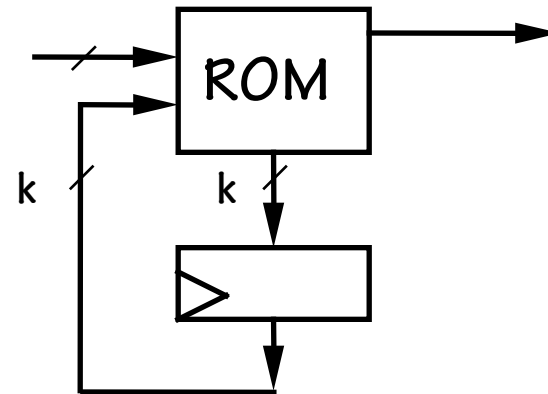
AMBIGUOUS TRANSITIONS (Mutual Exclusive property violated):
 For each input there can only be one arc leaving a state

UNSPECIFIED TRANSITIONS (Collectively Exhaustive property violated):
 There must be an arc leaving a state for all valid inputs (It can, however, loop back to the same state)

FSM Party Games

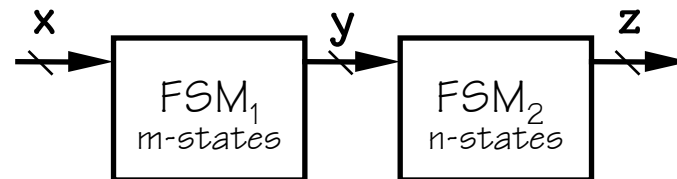
1. What can you say about the number of states?

$$\text{States} \leq 2^k$$

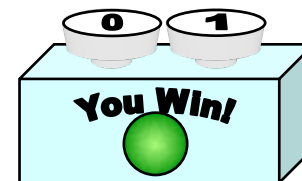


2. Same question:

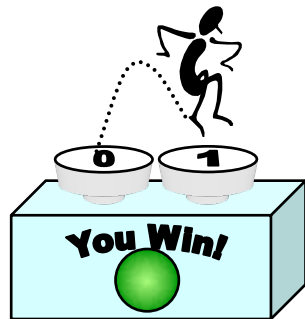
$$\text{States} \leq m \times n$$



3. Here's an FSM. Can you discover its rules?

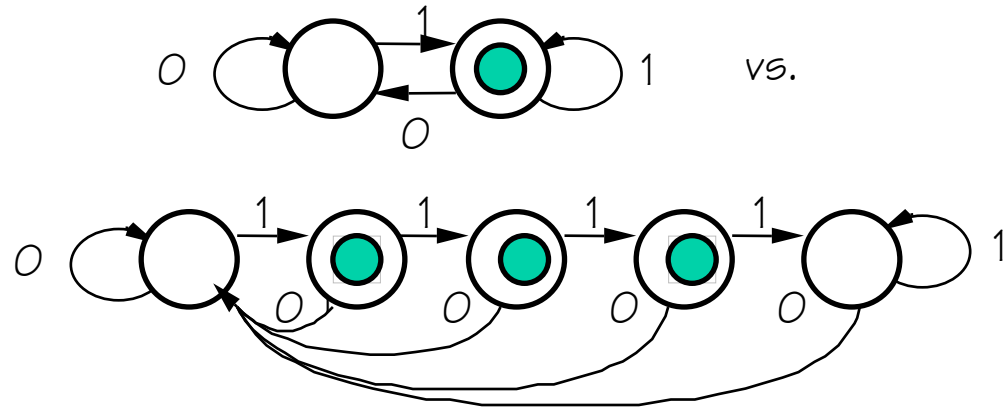


What's My Transition Diagram?



0=OFF,
1=ON?

"1111" =
Surprise!

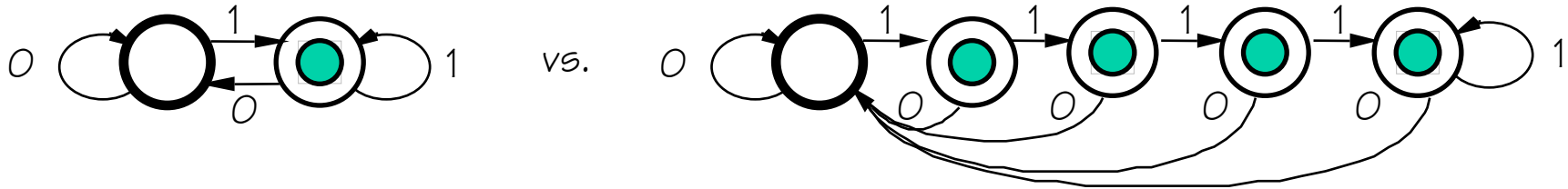


- If you know NOTHING about the FSM, you're never sure!
- If you have a BOUND on the number of states, you can discover its behavior:

K-state FSM: Every (reachable) state can be reached in $< 2^i \times k$ steps.

BUT ... states may be **equivalent!**

FSM Equivalence



ARE THEY DIFFERENT?

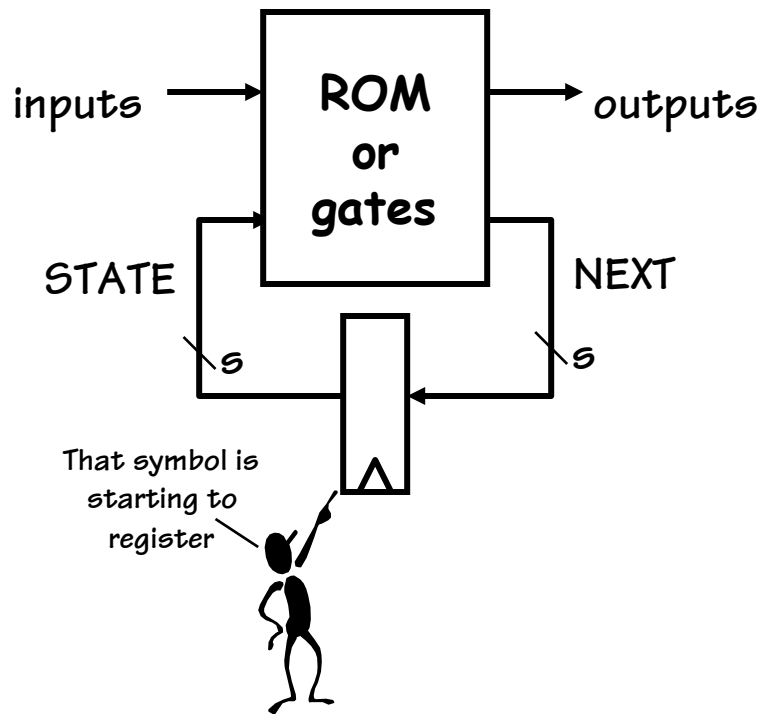
NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

FSMs are *EQUIVALENT* iff every input sequence yields identical output sequences.

ENGINEERING GOAL:

- HAVE an FSM which works...
- WANT simplest (ergo cheapest) equivalent FSM.

Housekeeping issues...

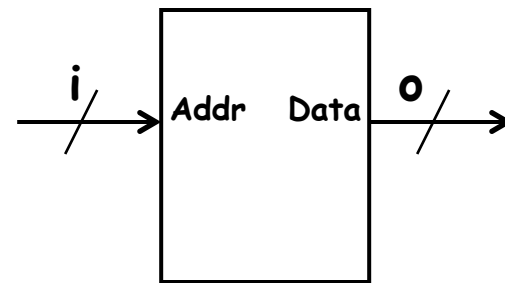


1. Initialization? Clear the memory?
2. Unused state encodings?
 - waste ROM (use gates)
 - meaning?
3. Synchronizing input changes with state update?
4. Choosing encoding for state?

2-Flavors of Processing Elements

Combinational Logic:
Table look-up, ROM

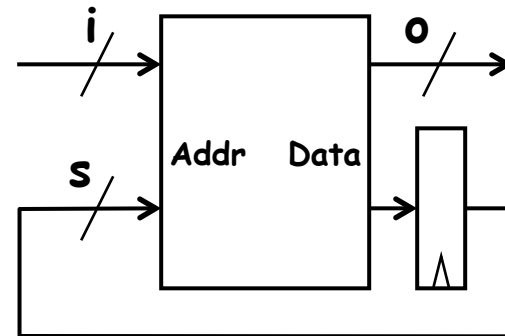
Recall that there are precisely 2^{2^i} , i -input combinational functions.
A single ROM can store 'o' of them.



Fundamentally,
everything
that we've
learned so far
can be done
with a ROM
and registers

Finite State Machines:
ROM with State Memory

Thus far, we know of nothing more
powerful than an FSM



FSMs as Programmable Machines

ROM-based FSM sketch:

An FSM's behavior is completely determined by its ROM contents.

Given i , s , and o ,
we need a ROM organized as:

2^{i+s} words \times $(o+s)$ bits

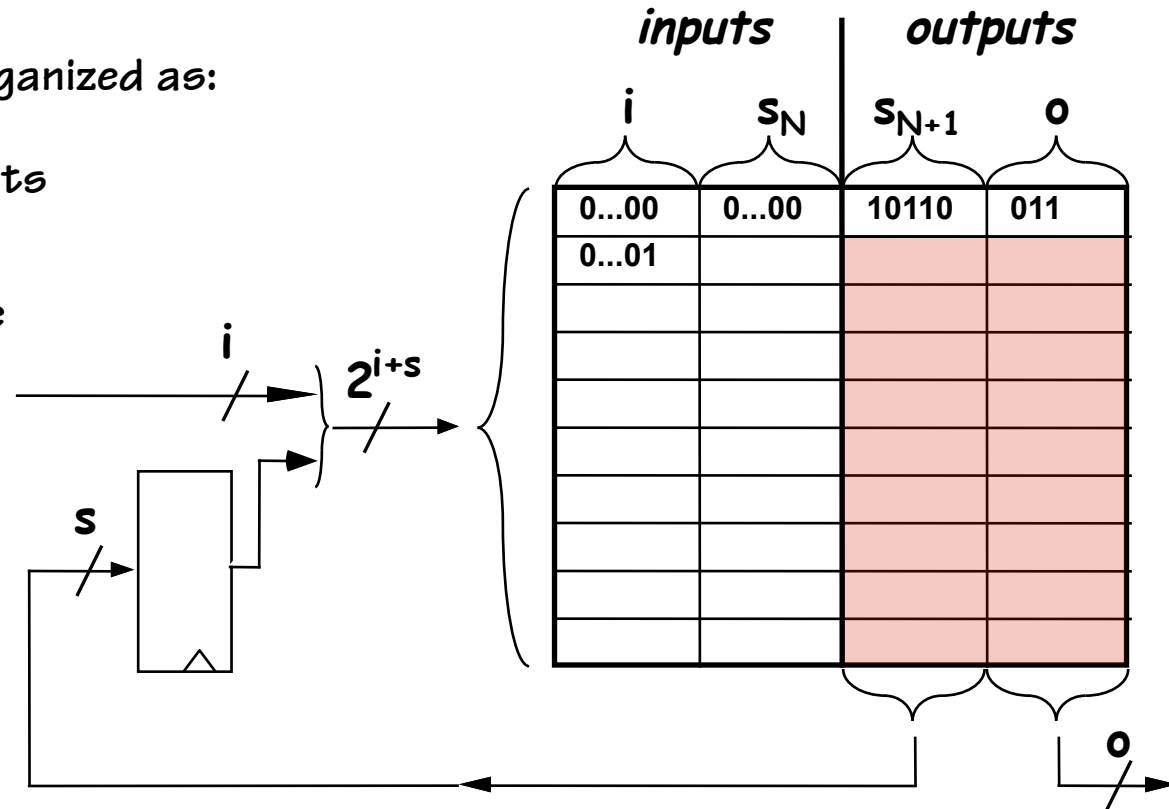
So how many possible
 i -input,
 o -output,
FSMs with
 s -state bits
exist?

All possible settings of the ROM's contents to: 1 or 0

$2^{(o+s)2^{i+s}}$

(some may be equivalent)

The number of "bits" in the ROM



Recall how we were able to "enumerate" or "name" every 2-input gate?
Can we do the same for FSMs?



How many state machines are there with 1-input, 1-output, and 1 state bit?

$$2^{(1+1)4} = 2^8 = 256$$

FSM Enumeration

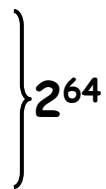
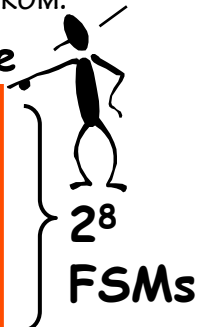
GOAL: List all possible FSMs in some canonical order.

- INFINITE list, but
- Every FSM has an entry in and an associated index.

inputs		outputs	
i	s _N	o	s _{N+1}
0...00	0...00	10110	011
0...01			

These are the FSMs with 1 input and 1 output and 1 state bit. They have 8-bits in their ROM.

i	s	o	FSM#	Truth Table
1	1	1	1	00000000
1	1	1	2	00000001
		
1	1	1	256	11111111
2	2	2	257	000000...000000
2	2	2	258	000000...000001
		
3	3	3	18,446,744,073,709,551,872 ...	000000...000000
		
4	4	4	3.9402 x 10 ¹¹⁵ ...	000000...000000



Every possible FSM can be associated with a unique number. This requires a few wasteful simplifications. First, given an i-input, s-state-bit, and o-output FSM, we'll replace it with its equivalent n-input, n-state-bit and n-output FSM, where n is the greatest of i, s, and o. We can always ignore the extra input-bits, and set the extra output bits to 0. This allows us to discuss the ith FSM

Some Perennial Favorites...

FSM₈₃₇

modulo 3 state machine

FSM₁₀₇₇

4-bit counter

FSM₁₅₃₇

Combination lock

FSM₈₉₁₄₃

Cheap digital watch

FSM₂₂₆₉₈₄₆₉₈₈₄

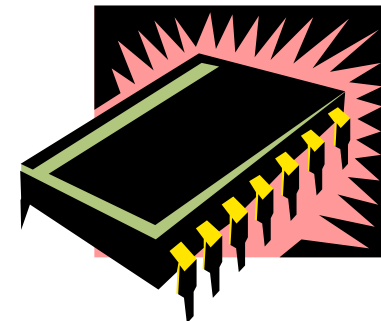
Intel Pentium CPU – rev 1

FSM₇₈₄₃₆₂₇₈₃

Intel Pentium CPU – rev 2

FSM₇₈₄₃₆₃₇₈₃

Intel Pentium II CPU



Can FSMs Compute Every Function?

Nope!

There exist many simple problems that cannot be computed by FSMs.

For instance:

Checking for balanced parenthesis

((()((()())))

- Okay

((()()))

- No good!



A function is specified by a deterministic output relationship for any given series of inputs, starting from a known initial state.

PROBLEM: Requires ARBITRARILY many states, depending on input. Must "COUNT" unmatched LEFT parens.

But, an FSM can only keep track of a "bounded" number of events. (Bounded by its number of states)

Is there a machine that can solve this problem?

Unbounded-Space Computation

DURING 1920s & 1930s, much of the “science” part of computer science was being developed (long before actual electronic computers existed). Many different

“Models of Computation”

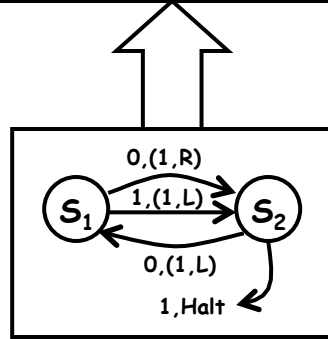
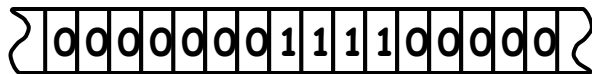
were proposed, and the classes of “functions” that each could compute were analyzed.

One of these models was the

“TURING MACHINE”,

named after Alan Turing (1912-1954).

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an “infinite tape”. This simple addition overcomes the FSM’s limitation that it can only keep track of a “bounded number of events”.



Alan Turing

A Turing Machine Example

Turing Machine Specification

- Infinite tape
- Discrete symbol positions
- Finite alphabet – say {0, 1}
- Control FSM

INPUTS:

Current symbol on tape

OUTPUTS:

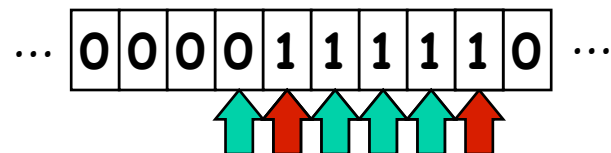
write 0/1

move Left/Right

- Initial Starting State {S0}
- Halt State {Halt}

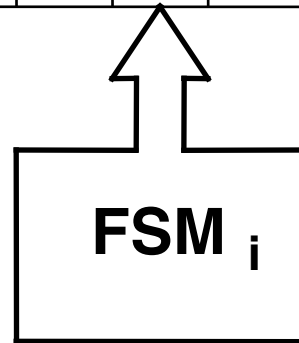
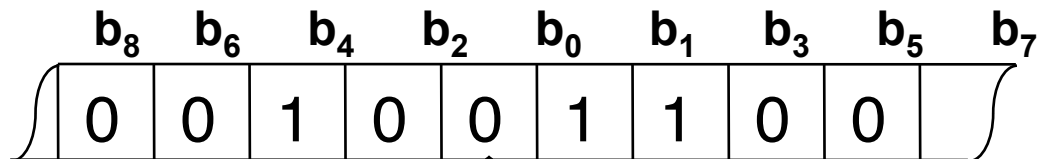
A Turing machine, like an FSM, can be specified via a state-transition table. The following Turing Machine implements a unary (base 1) incrementer.

Current State	Tape Input	Write Tape	Move	Next State
S0	1	1	R	S0
S0	0	1	L	S1
S1	1	1	L	S1
S1	0	0	R	Halt



Turing Machine Tapes as Integers

Canonical names for bounded tape configurations:



Look, it's just FSM i operating on tape j



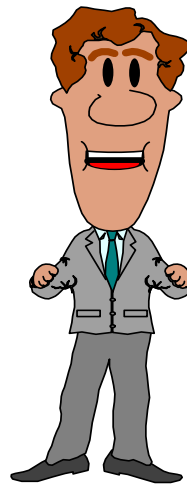
Note: The FSM part of a Turing Machine is just one of the FSMs in our enumeration. The tape can also be represented as an integer, but this is trickier. It is natural to represent it as a binary fraction, with a binary point just to the left of the starting position. If the binary number is rational, we can alternate bits from each side of the binary point until all that is left is zeros, then we have an integer.

TMs as Integer Functions

Turing Machine T_i operating on Tape x ,
where $x = \dots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

$$y = T_i [x]$$

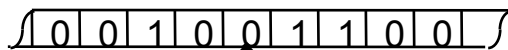
x : input tape configuration
 y : output tape when TM *halts*



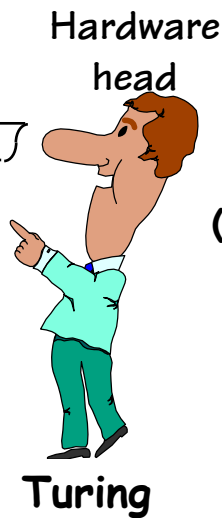
I wonder if a TM can compute
EVERY integer function...

Alternative Models of Computation

Turing Machines [Turing]



FSM_i



Turing

Recursive Functions [Kleene]

$$F(0,x) \equiv x$$

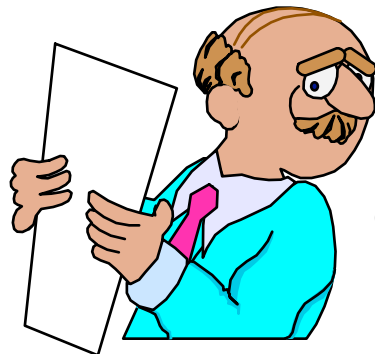
$$F(1+y,x) \equiv 1+F(x,y)$$

```
(define (fact n)
 (... (fact (- n 1)) ...))
```



Kleene (1909-1994)

Lambda calculus [Church, Curry, Rosser...]



Math
head

$$\lambda x. \lambda y. xxy$$

```
(lambda (x) (lambda (y) (x (x y))))
```

Church (1903-1995)
Turing's PhD Advisor

Production Systems [Post, Markov]

Language
head

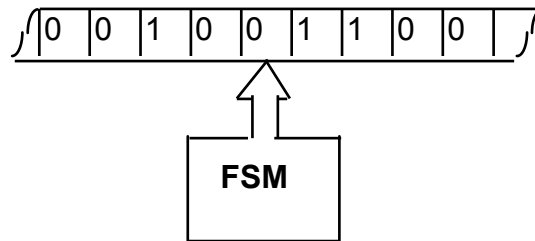
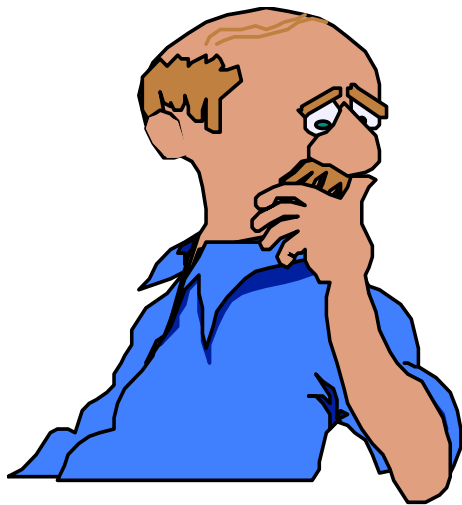


$$\begin{aligned} \$_0 &\rightarrow [] \\ \$ &\rightarrow [\$] \\ \$ &\rightarrow \$\$ \\ \$_i [] \$_j &\rightarrow \$_i \$_j \end{aligned}$$

Post
(1897-1954)

The 1st Computer Industry Shakeout

Here's a TM that computes SQUARE ROOT!



And the Battles Raged

Here's a Lambda Expression
that does the same thing...

$(\lambda (x) \dots\dots)$

... and here's one that computes
the n^{th} root for ANY $n!$

$(\lambda (x \ n) \dots\dots)$



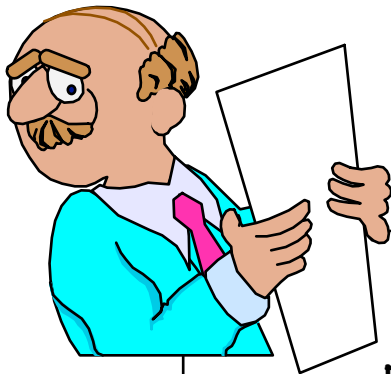
Fundamental Result: Computable Functions

Each model is capable of computing exactly the same set of integer functions!

Proof Technique: Constructions that translate between models

BIG IDEA: Computability, independent of computation scheme chosen

This means that we know of no mechanisms (including computers) that are more "powerful" than a Turing Machine, in terms of the functions they can compute.



Church's Thesis:

Every discrete function computable by ANY realizable machine is computable by some Turing machine.



Computable Functions

The “input” to our computable function will be given on the initial tape, and the “output” will be the contents of the tape when the TM halts.

$f(x)$ *computable* \Leftrightarrow for some k , all x :

$$f(x) = T_k[x] \equiv f_k(x)$$



Representation tricks: to compute $f_k(x,y)$ (2 inputs)

$\langle x,y \rangle \equiv$ integer whose *even* bits come from x ,
and whose *odd* bits come from y ; whence

$$f_k(x, y) \equiv T_k[\langle x, y \rangle]$$

$$f_{12345}(x,y) = x * y$$

$$f_{23456}(x) = 1 \text{ iff } x \text{ is prime, else } 0$$

TMs, like programs, can misbehave

It is possible that a given Turing Machine may not produce a result for a given input tape. And it may do so by entering an infinite loop!

Consider the given TM.

Its scans a tape looking for the first non-zero cell to the right.

What does it do when given a tape that has no 1's to its left?

We say this TM does not halt for that input!

Current State	Tape Input	Write Tape	Move	Next State
S0	1	1	L	Halt
S0	0	0	R	S0

tape₂₅₆ = ... 0 0 0 0 0 0 0 0 1 0 0 ...



tape₈ = ... 0 1 0 0 0 0 0 0 0 0 0 ...



Enumeration of Computable functions

Conceptual table of TM behaviors...

VERTICAL AXIS: Enumeration of TMs.

HORIZONTAL AXIS: Enumeration of input tapes.

(j, k) entry = result of $TM_k[j]$ -- integer, or * if never halts.

Turing Machine Tapes \longrightarrow

f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$...	$f_i(j)$...
f_0	3 1	2 1	* 0	...		
f_1	6 1	* 0		
...		
f_k	$f_k(j)$	
...	

Turing Machine FSMs \downarrow

Every computable function is in this table, since everything that we know how to compute can be computed by a TM.

Do there exist well-specified integer functions that a TM can't compute?



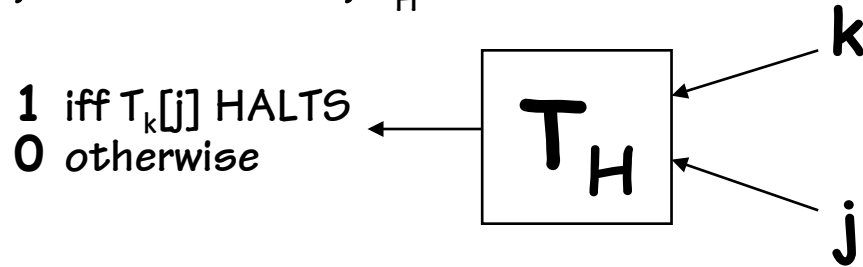
The Halting Problem: Given j, k: Does TM_k Halt with input j?

The Halting Problem

The Halting Function: $T_H[k, j] = 1$ iff $TM_k[j]$ halts, else 0

Can a Turing machine compute this function?

Suppose, for a moment, T_H exists:



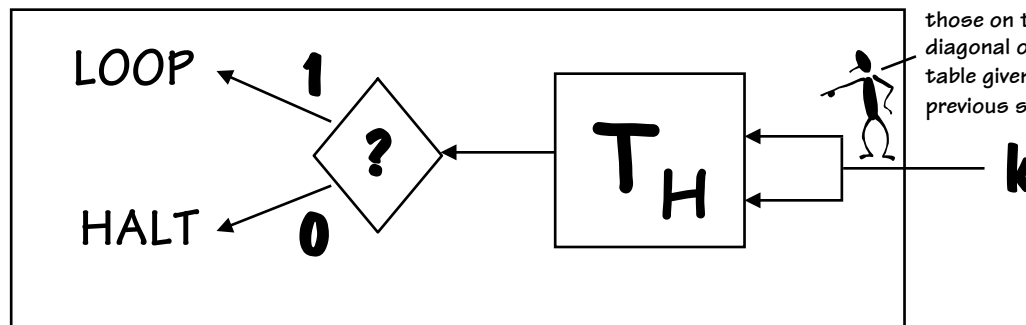
If T_H is computable then so is

T_{Nasty}



We only run T_H on a subset of inputs, those on the diagonal of the table given on the previous slide

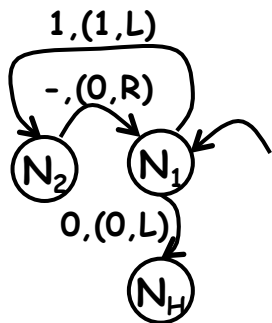
Then we can build a T_{Nasty} :



$T_{Nasty}[k]$

LOOP if $T_k[k] = 1$ (halts)
 HALT if $T_k[k] = 0$ (loops)

Replace the Halt state of TH with this.



What does $T_{\text{Nasty}}[\text{Nasty}]$ do?

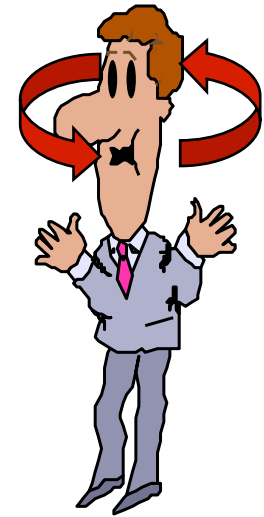
Answer:

$T_{\text{Nasty}}[\text{Nasty}]$ loops if $T_{\text{Nasty}}[\text{Nasty}]$ halts

$T_{\text{Nasty}}[\text{Nasty}]$ halts if $T_{\text{Nasty}}[\text{Nasty}]$ loops

That's a contradiction.

Thus, T_H is uncomputable by a Turing Machine!



Net Result: There are some integer functions that Turing Machines simply cannot answer. Since, we know of no better model of computation than a Turing machine, this implies that there are some well-specified problems that defy computation.

Reality: Limits of Turing Machines

A Turing machine is formal abstraction that addresses

- Fundamental Limits of Computability –
What it means to compute.
The existence of incomputable functions.
- We know of no machine more powerful than a Turing machine in terms of the functions that it can compute.

But they ignore

- Practical coding of programs
- Performance
- Implementability
- Programmability

... these latter issues are the primary focus of contemporary computer science (Remainder of Comp 411)

Computability vs. Programmability

Recall Church's thesis:

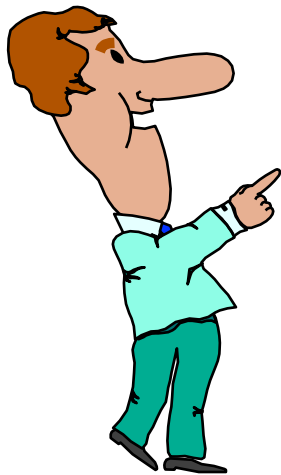
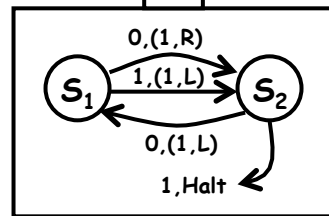
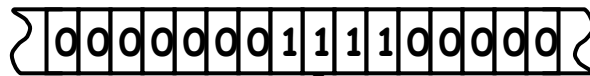
"Any discrete function computable by ANY realizable machine is computable by some Turing Machine"

Thus, we've defined what it means to COMPUTE (whatever a TM can compute)

A Turing machine is nothing more than an FSM that receives inputs from, and outputs onto, an infinite tape.

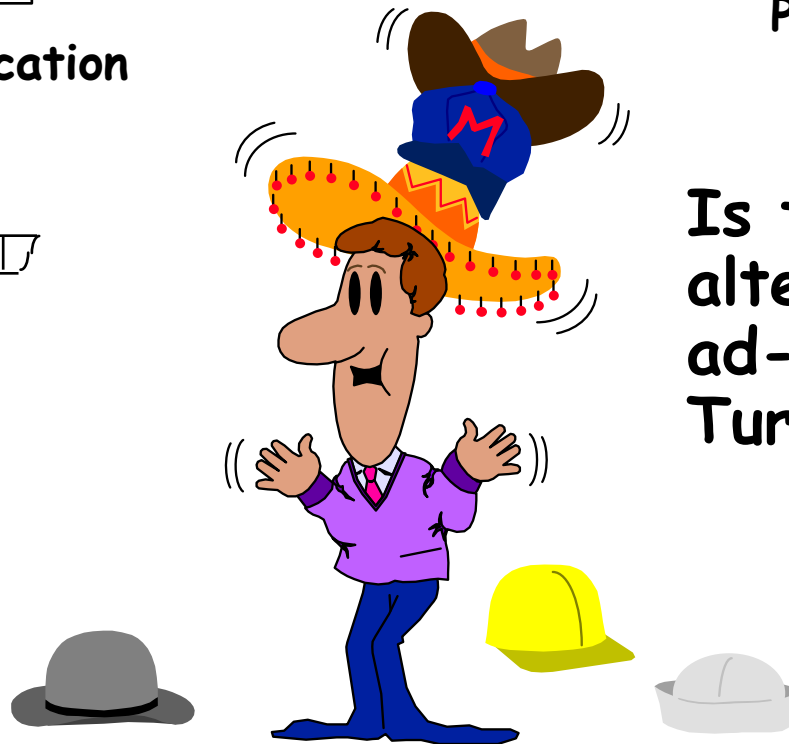
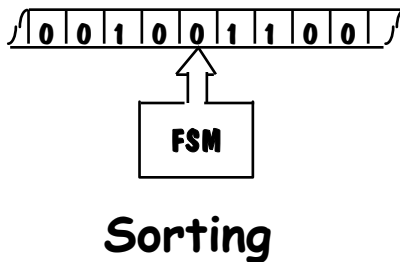
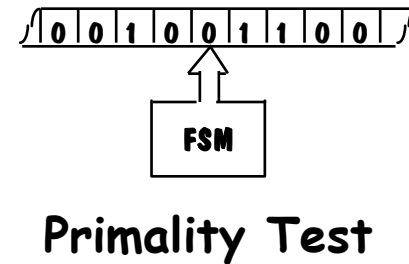
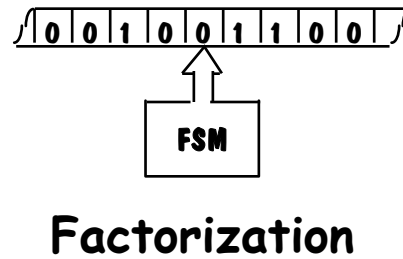
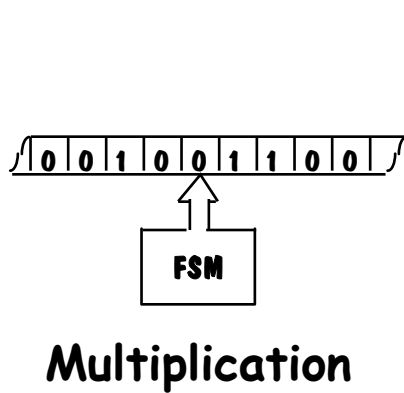
Thus far, we've been designing a new Turing machine FSM for each new function that we encounter.

Wouldn't it be nice if we could design a more general purpose computing machine?



Alan Turing

Too many Turing machines!

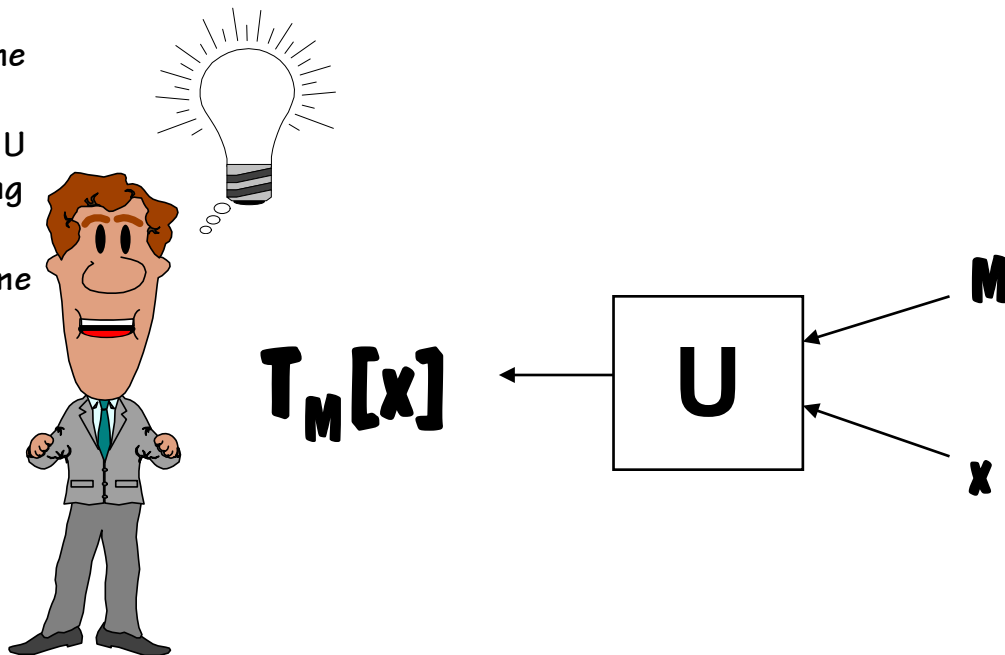


Is there an
alternative to
ad-hoc
Turing Machines?

Programs as Data

What if we encoded the description of the FSM on our tape, and then wrote a general purpose FSM to read the tape and *EMULATE* the behavior of the encoded machine? We could just store the state-transition table for our TM on the tape and then design a new TM that makes reference to it as often as it likes. It seems possible that such a machine could be built.

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M."
- Turing 1936 (Proc of the London Mathematical Society, Ser. 2, Vol. 42)



Fundamental Result: Universality

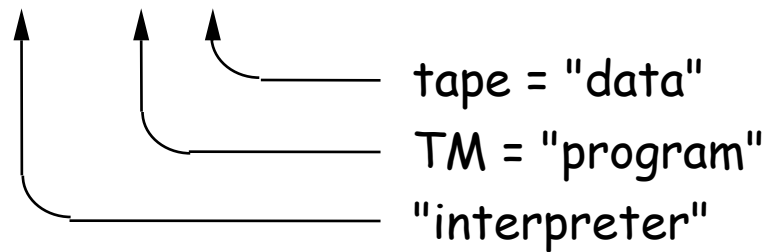
Define "Universal Function": $U(x,y) = T_x(y)$ for every $x, y \dots$

Surprise! $U(x,y)$ IS COMPUTABLE,

hence $U(x,y) = T_U(\langle x,y \rangle)$ for some U .

Universal Turing Machine (UTM):

$$T_U[\langle y, z \rangle] = T_y[z]$$



PARADIGM for General-Purpose Computer!

INFINITELY many UTMs ...

Any one of them can evaluate any computable function by simulating/emulating/interpreting the actions of Turing machine given to it as an input.

UNIVERSALITY:

Basic requirement for a general purpose computer

Demonstrating Universality

Suppose you've designed Turing Machine T_K and want to show that its universal.

APPROACH:

1. Find some known universal machine, say T_U .
2. Devise a program, P , to simulate T_U on T_K :
 $T_K[\langle P, x \rangle] = T_U[x]$ for all x .
3. Since $T_U[\langle y, z \rangle] = T_Y[z]$, it follows that, for all y and z .

$$T_K [\langle P, \langle y, z \rangle \rangle] = T_U[\langle y, z \rangle] = T_Y[z]$$

CONCLUSION: Armed with program P , machine T_K can mimic the behavior of an arbitrary machine T_Y operating on an arbitrary input tape z .

HENCE T_K can compute any function that can be computed by any Turing Machine.

Turing Complete

Interpretive Layers: What's going on?

$$T_K [\langle P, \langle y, z \rangle \rangle] = T_U [\langle y, z \rangle] = T_Y [z]$$

Multiple levels of interpretation:

$T_Y [z]$	Application (Desired user function)
$T_U [\langle y, z \rangle]$	Portable Language / Virtual Machine
$T_K [\langle P, \langle y, z \rangle \rangle]$	Computing Hardware / Bare Metal

Benefits of Interpretation:

BOOTSTRAP high-level functionality on very simple hardware.

Deal with “**IDEAL**” machines rather than real machines.

REAL MACHINES are built this way - several interpretive layers.

Power of Interpretation

BIG IDEA: Manipulate *coded representations* of computing machines, rather than the machines themselves.

- PROGRAM as a behavioral description
- SOFTWARE vs. HARDWARE
- INTERPRETER as machine which takes program and mimics behavior it describes
- LANGUAGE as interface between interpreter and program
- COMPILER as translator between languages:

INTELLECTUAL BENEFITS:

- Programs as data -- mathematical objects
- Combination, composition, generation, parameterization, etc.