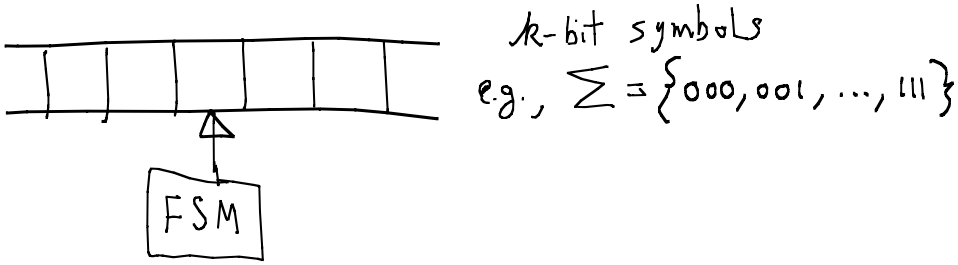
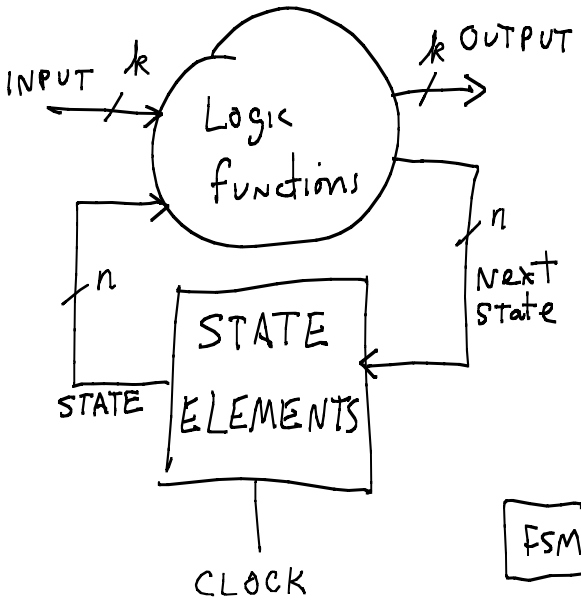
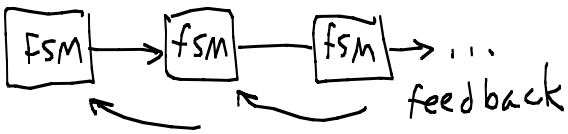
FSM-implementation



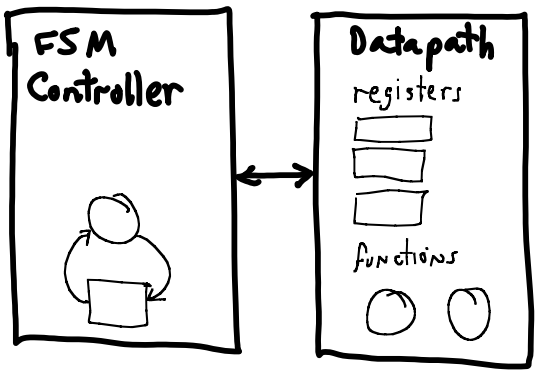




FSM has input/output, but from/to where?

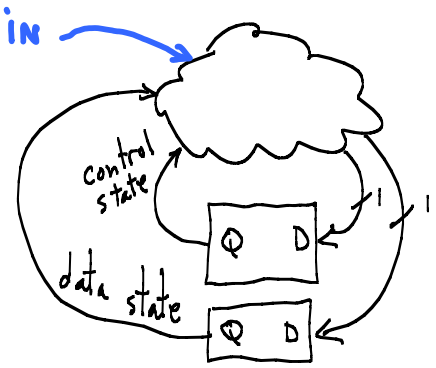
(1) Other FSMs in same machine

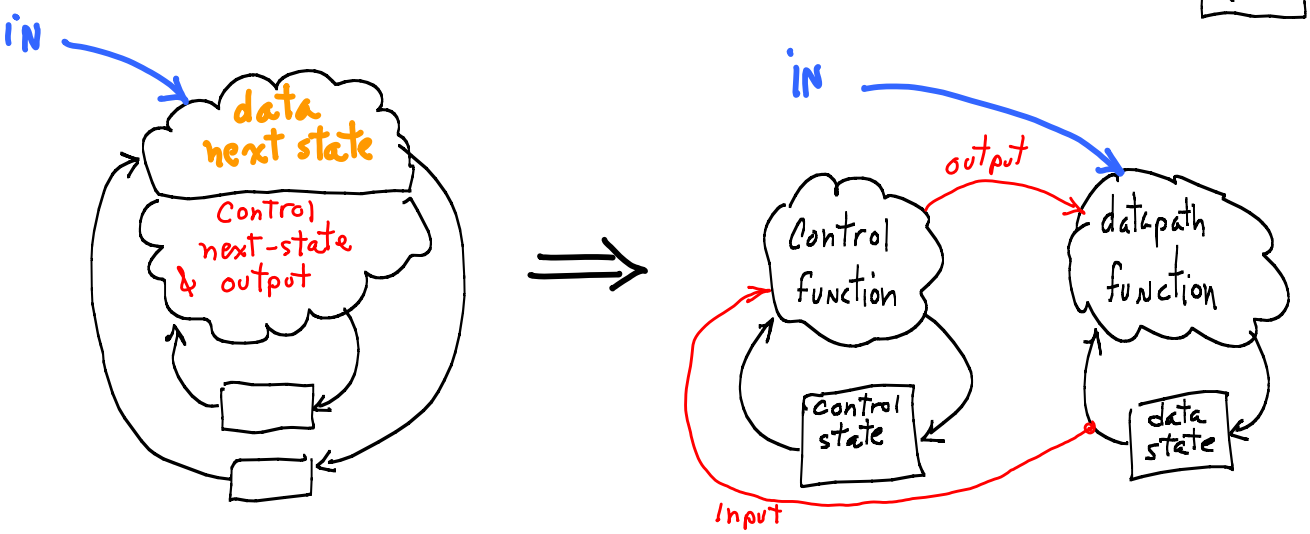
(2) Feedback loops

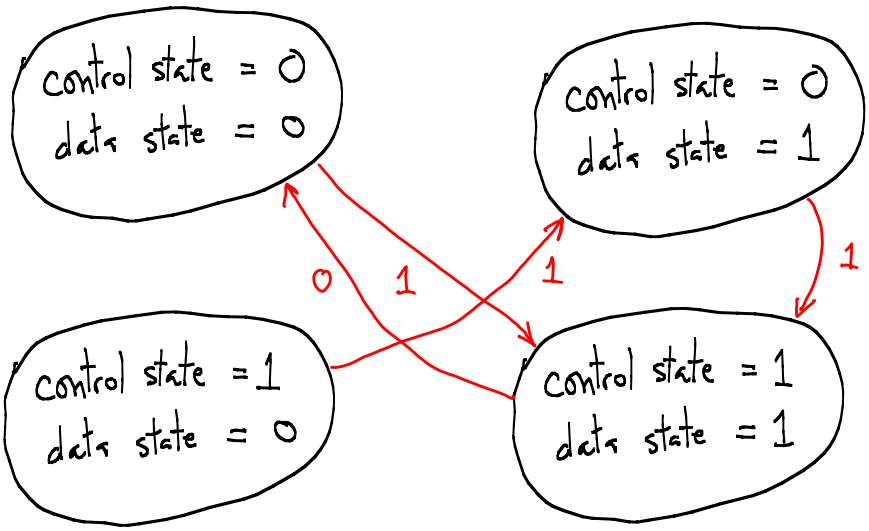


We like to separate state into two "types", control and data state. Eg., some state elements are for "control" state, and some are for "data" state.

eg.

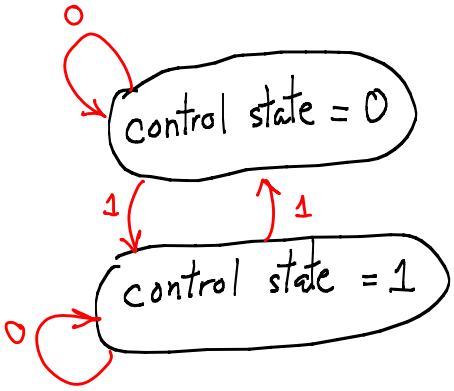


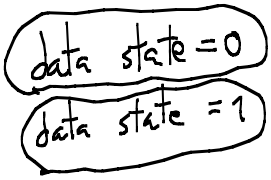




Total machine state

Suppose a FSM with 1-bit control-state element and 1-bit data-state element. Total is 2-bits of state; 4-state machine. State diagram quickly becomes a mess. What if 2-bit data-state? 8-state total FSM, but only 2 control states.





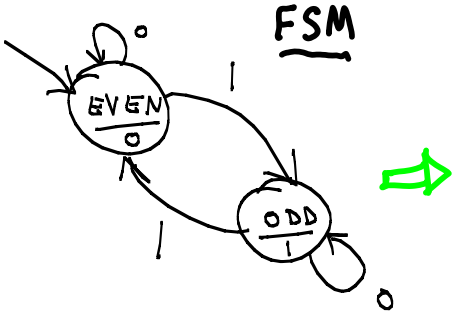
Splitting state: "control state" "datapath state"

In any control state, data state can be in either of two states: 0 or 1. Data states could be much larger: using just one 32-bit data register gives us 4 G data states. With this 2-state control, total machine state would be 8 G states.

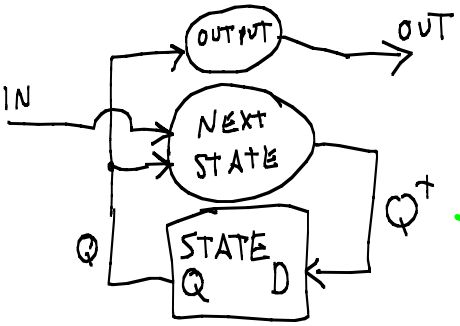
FSM-circuits, implementation example

Serial parity bit machine

FSM



Moore Machine



STATE ENCODING	
EVEN	Q = 0
ODD	Q = 1

or

EVEN	$Q_1 = 1$
ODD	$Q_1 = 0$

OR

EVEN	$Q_0 = 1$ $Q_1 = 0$
ODD	$Q_0 = 0$ $Q_1 = 1$

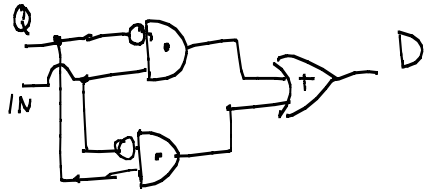
Next-State function

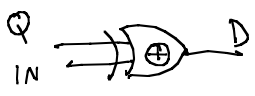
Q	IN	$D = Q^+$	
0	0	0	
0	1	1	$= m_1$
1	0	1	$= m_2$
1	1	0	

$D = \overline{Q} \cdot IN + Q \cdot \overline{IN}$

$= Q \oplus IN$

Next-state circuit



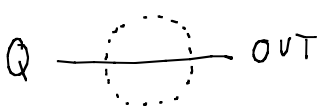


OUTPUT FUNCTION

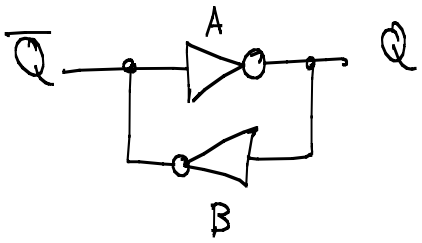
Q	OUT
0	0
1	1

OUT = Q

OUTPUT circuit



STATE Elements



Suppose A.out = 0
→ B.out = 1
→ A.out = 0
STable!

A.out = 1
→ B.out = 0
→ A.out = 1
STable

That was easy, but Hey! No inputs! :c°

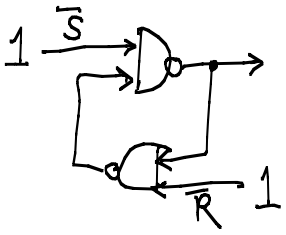
Q = 0 | Q = 1

two states! hooray!

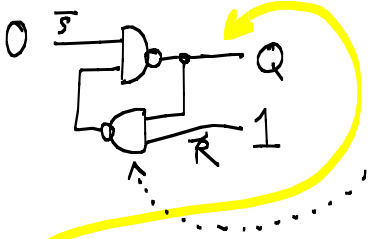
X	Y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

hmm, if X=1, looks like NOT

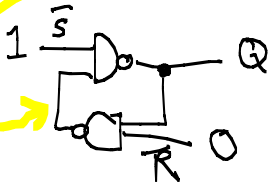
hmm, if X=0, always gives 1



can we use inputs?



forces 1, this is still a NOT



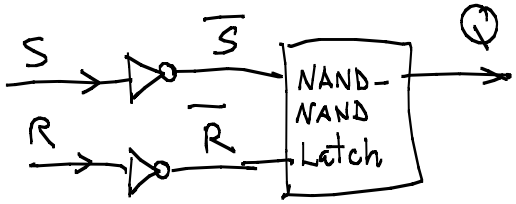
forces 1, now in other stable state!

The state is latched (captured) when -S = -R = 1.
NAND-NAND latch

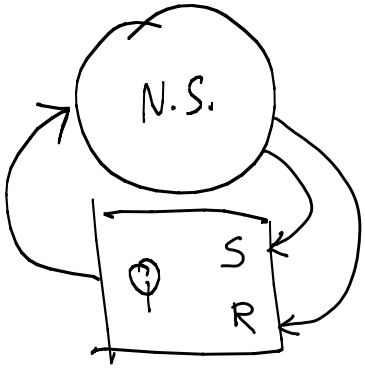
$\overline{S}$	$\overline{R}$	STATE
1	1	INV-INV Loop, 2 possible stable states: Q=0 or Q=1
0	1	1-INV Loop, 1 possible state: Q=1
1	1	INV-INV Loop, latched state Q=1
1	0	INV-1 Loop, one possible state: Q=0
1	1	INV-INV Loop, Latched state Q=0

Yay! State element w/ input!

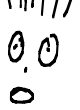
S-R Latch



Can we use this for a FSM state element?



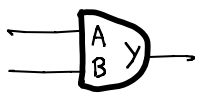
Oh No!



S = set
R = reset

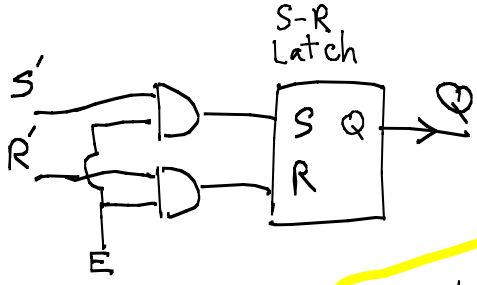
any input change → to output → state change!

CLOCKING, part 1: Gating D-Latch



B = 0 → Y = 0

B = 1 → Y = A



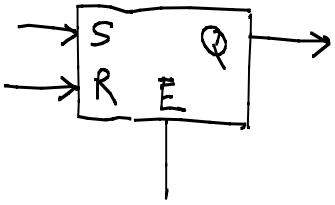
E	A	B	Y = A·B
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

→ S = R = 0

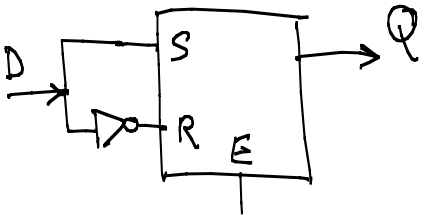
Enable = 0, input is ignored

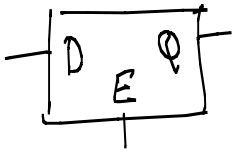
Enable = 1, input allowed through: R = R', S = S'

S-R latch w/ Enable



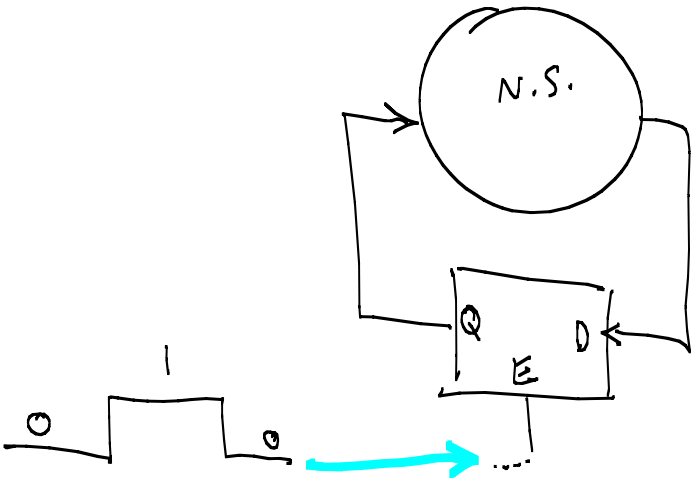
conceptually simplify ⇒



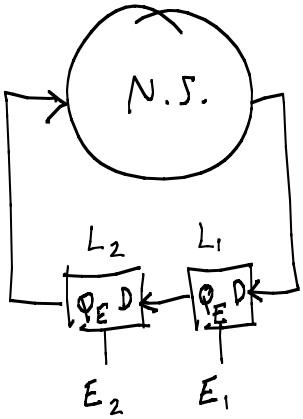


D-latch w/ Enable

2-phase clocking, D-FF

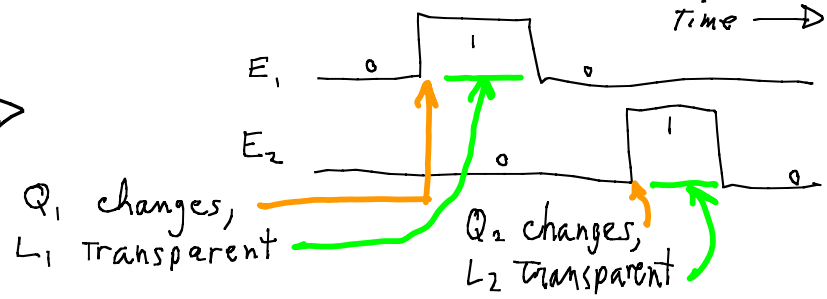


⇒

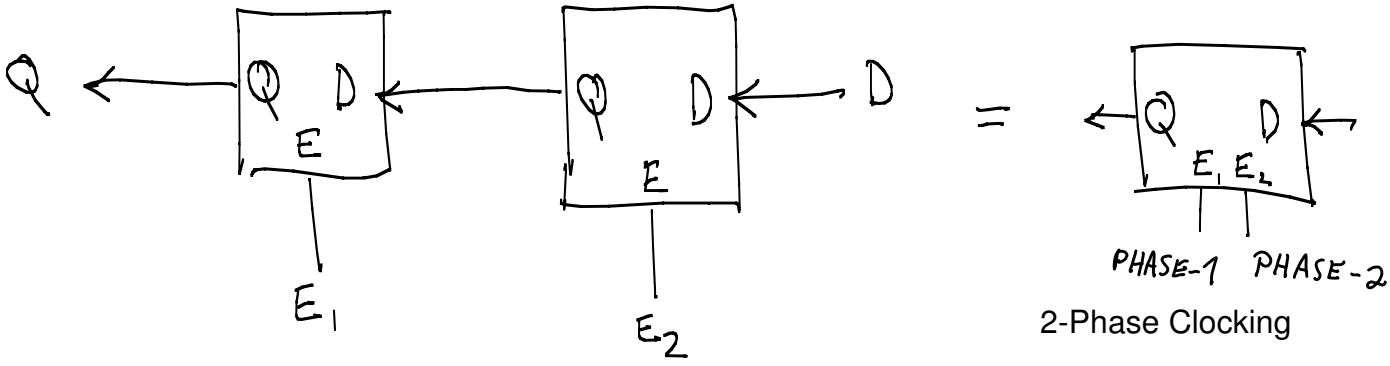


Transparent during this time, enough time to cause state change?

Make sure system never has an open feedback path. ⇒

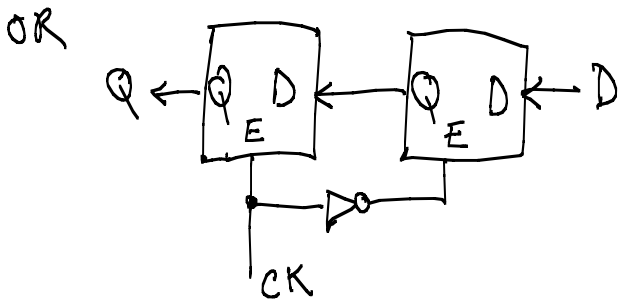


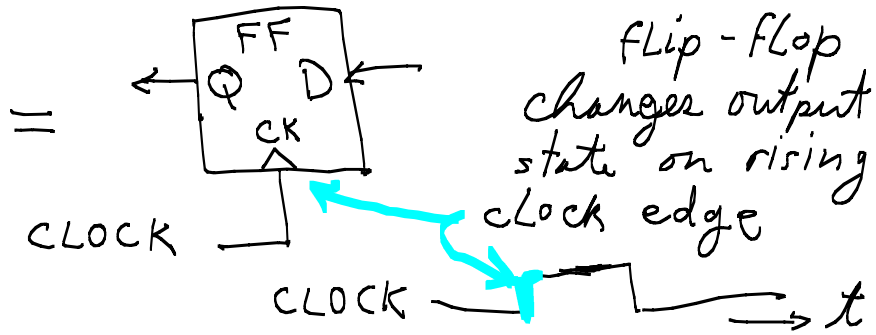
D-FF



2-Phase Clocking

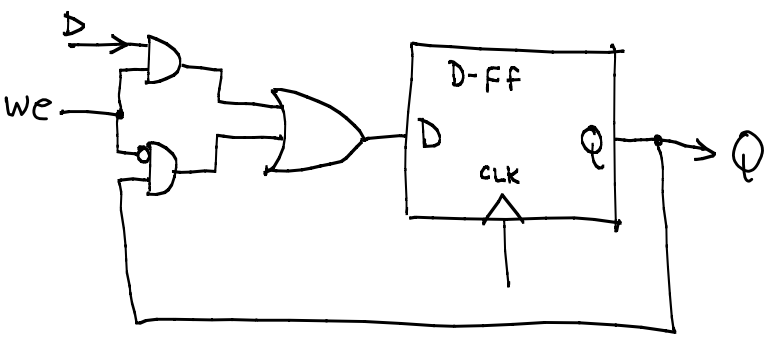
Separate signals for each latch's enable in FlipFlop. On breadboard we connect PHASE-1 to one data switch, PHASE-2 to another.

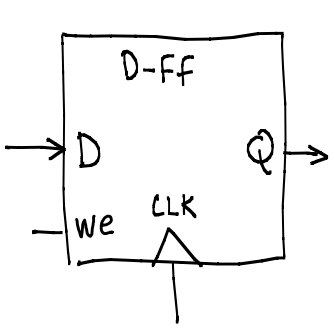




D-FF, Pos. edge triggered w/enable

We often want to control whether or not the FF will be written into when the clock pulse arrives: add an "enable" input. When enable is 0, the current state is written back into the FF. Otherwise, D is written.

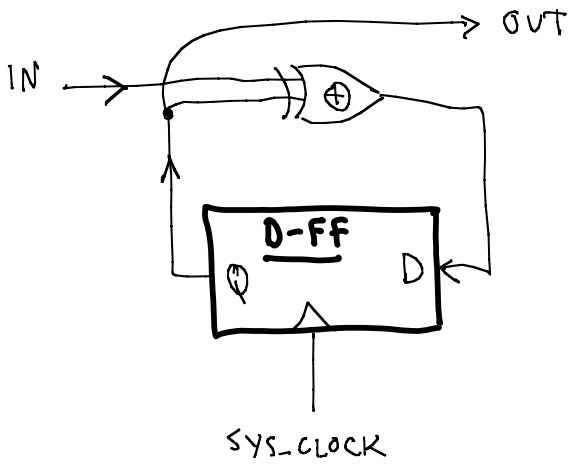




D-FF w/ write enable

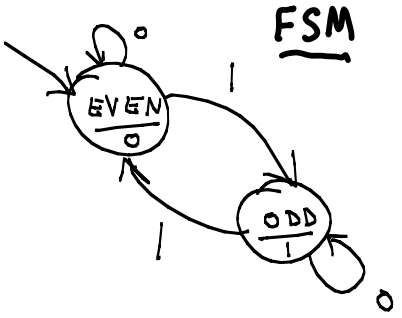
If there is no feedback path from Q to D, we do not need a flip-flop, we can use a write-enable latch instead. Datapaths sometimes can use latches.

Serial Parity, final implementation



This is a FSM with non-trivial control state and next-state function, but a trivial datapath.

Flipflop can be 2-phase clocked, in which case the "sys_clock" signal consists of two wires.



STATE ENCODING	
EVEN	Q = 0
ODD	Q = 1

next-state

$D = IN \oplus Q$

output

OUT = Q

FSM implementation, example | ADD-2, 2 serial inputs \ reg.

$C_4\ C_3\ C_2\ C_1\ C_0$

$X_3\ X_2\ X_1\ X_0$

$+\ Y_3\ Y_2\ Y_1\ Y_0$

$S_4\ S_3\ S_2\ S_1\ S_0$

$\ldots\ X_3\ X_2\ X_1\ X_0 \rightarrow X_{in}$

$\ldots\ Y_3\ Y_2\ Y_1\ Y_0 \rightarrow Y_{in}$

Time ←

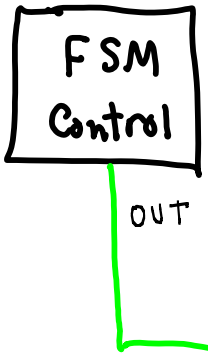
$S_{out} \rightarrow \ldots\ S_2 S_1 S_0$

FSM

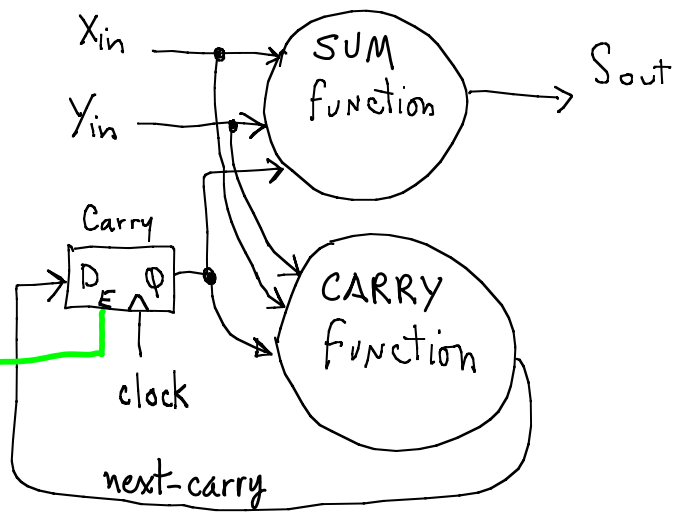
state encoding: S0 = 1

next-state function: Q+ = Q

output function: out = Q



data path



Datapath functions, MUX

C_i

X_i

$+\ Y_i$

$C_{i+1}\ S_i$

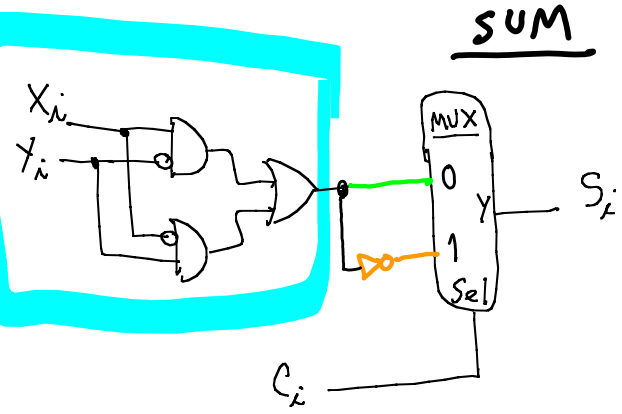
C_i	Y_i	X_i	S_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$C_i = 0$

$C_i = 1$

NOT

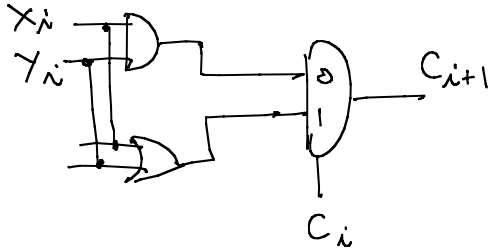
SUM



C_i	Y_i	X_i	C_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

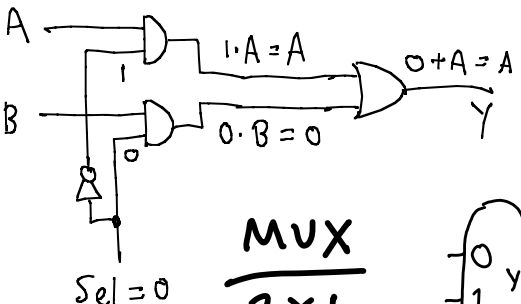
AND

OR

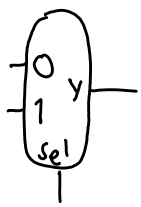


MAJORITY

Select one of two inputs

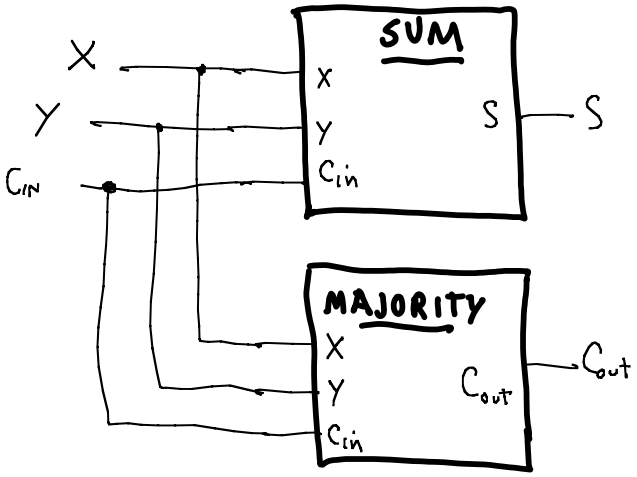


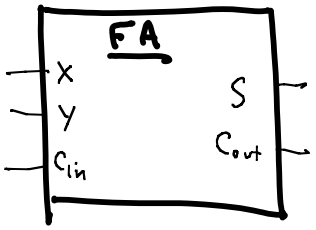
MUX 2×1



Full Adder

FA

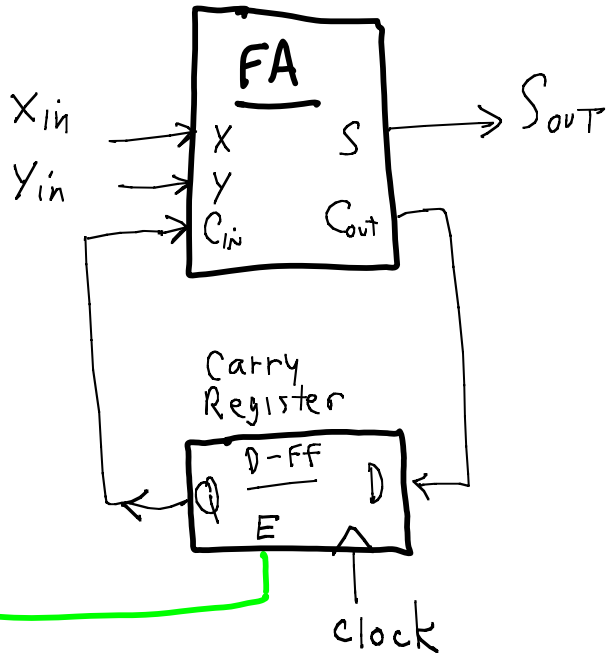




Add-2, final implementation

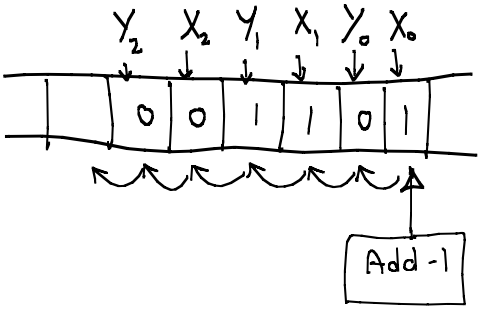
data path

This FSM has a non-trivial datapath consisting of datapath "state" registers and data processing "next-state" functions. But the control FSM is trivial.



FSM control

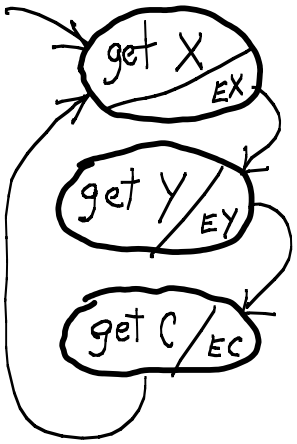
FSM, Add-1 implementation



$$\begin{array}{r} C_2\ C_1\ C_0 \\ X = X_2 X_1 X_0 \\ +\ Y = Y_2 Y_1 Y_0 \\ \hline S = \ldots S_2 S_1 S_0 \end{array}$$

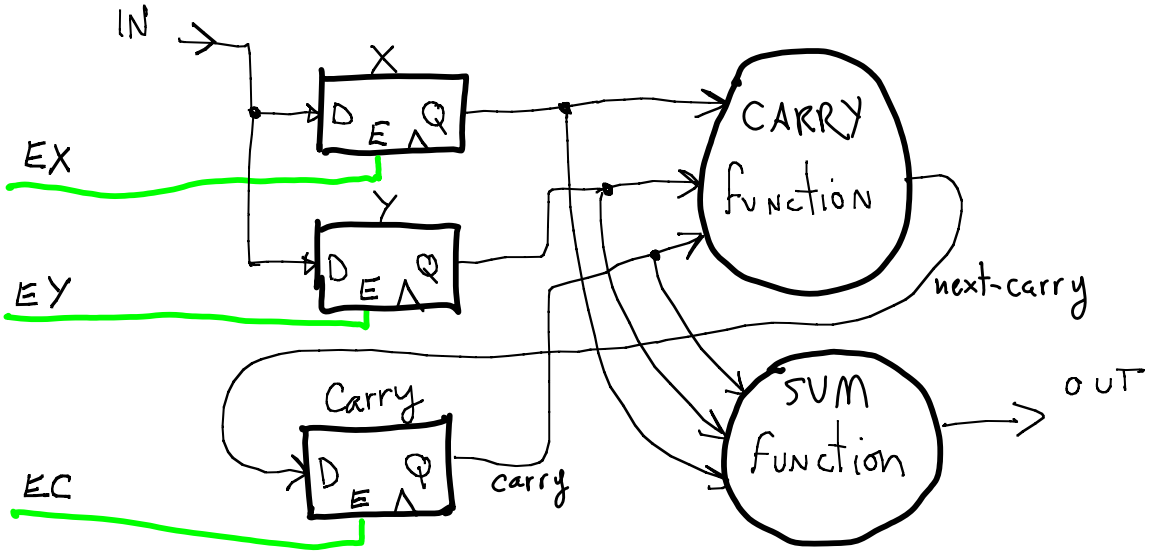
And Now, a FSM with a non-trivial controller and non-trivial datapath.

FSM control

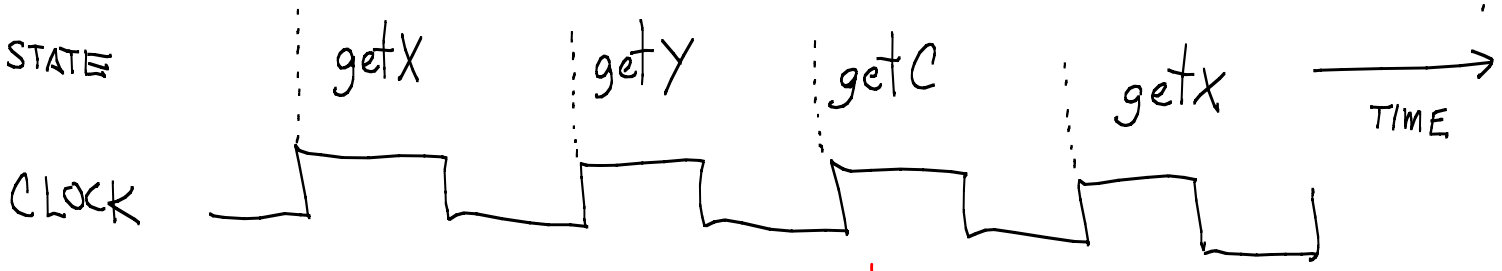


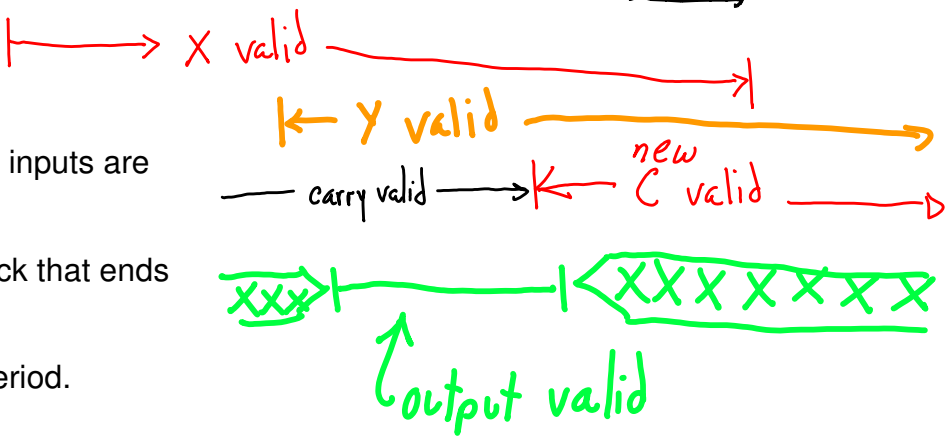
⊘x means x=1, all others = 0

data Path



When is output valid? When should inputs be valid?





Function outputs are not valid until all inputs are available (valid).

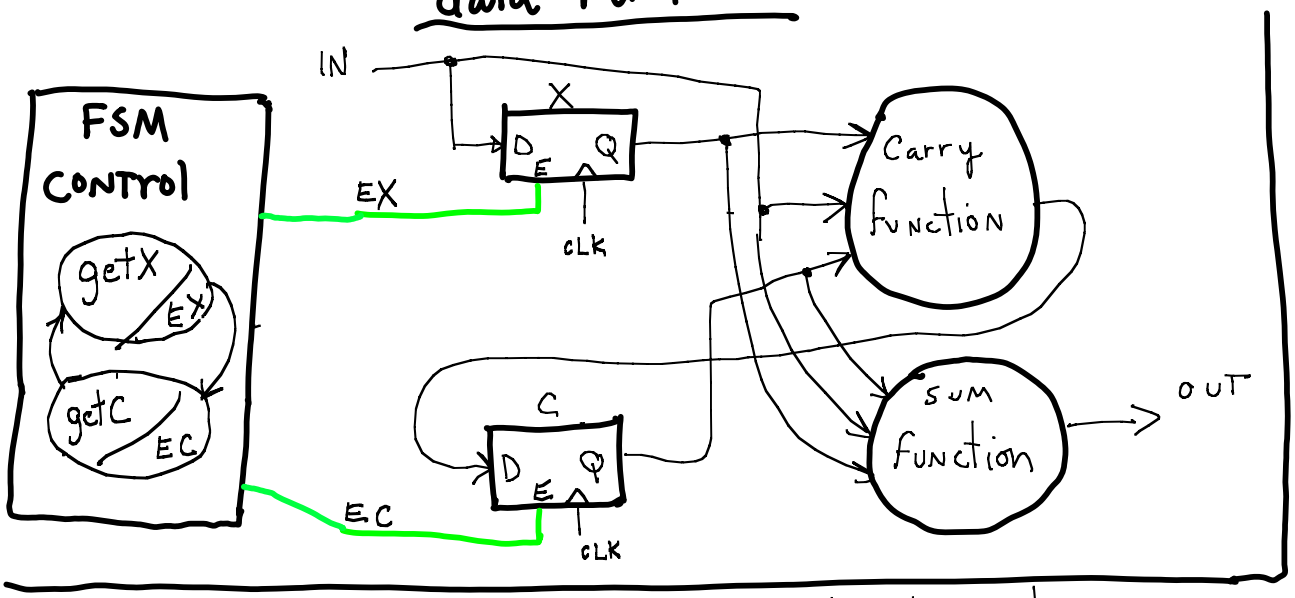
X does not become valid until clock tick that ends state getX.

Output is only valid every 3rd clock period.

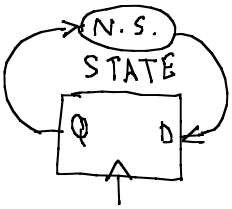
Simplified FSM and datapath

If we can depend on the input to be stable, we can eliminate the Y data register: just use the data input when it is valid for Y. Eliminates one control state.

data Path



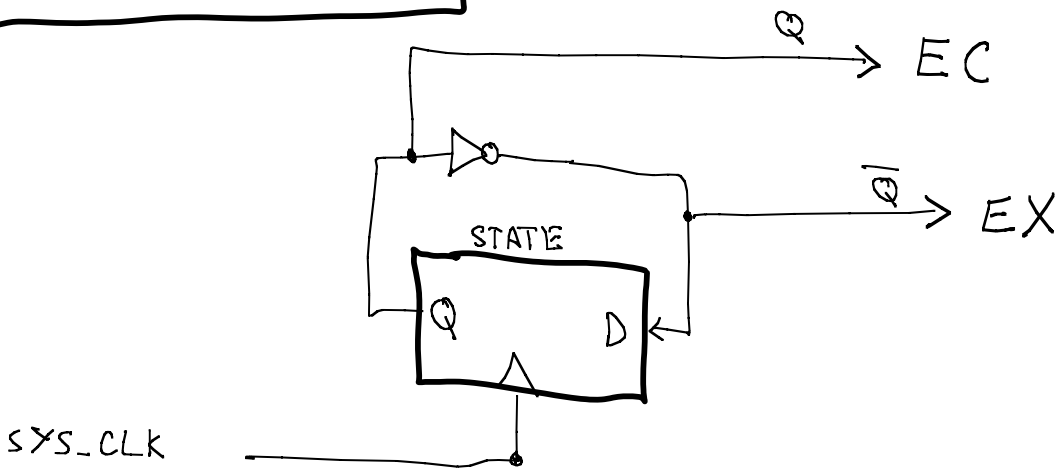
STATE	ENCODING
getX	0
getC	1



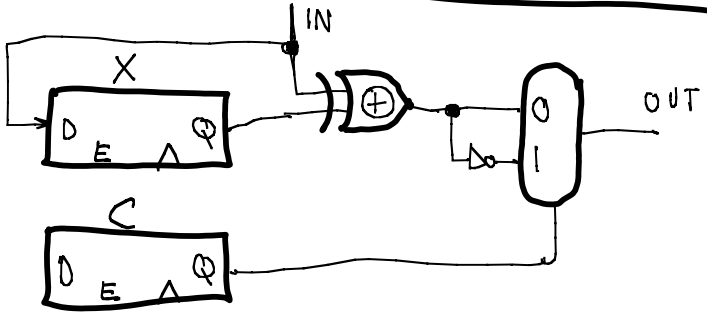
Next-State function

STATE.Q	STATE.D = Q^+
0	1
1	0

FSM controller



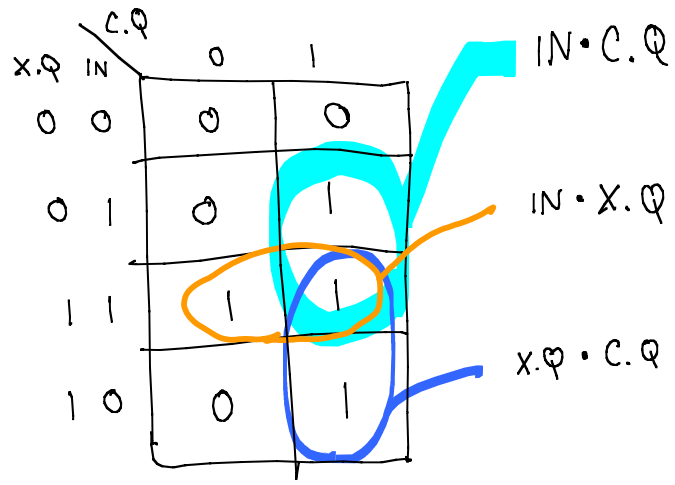
Sum function

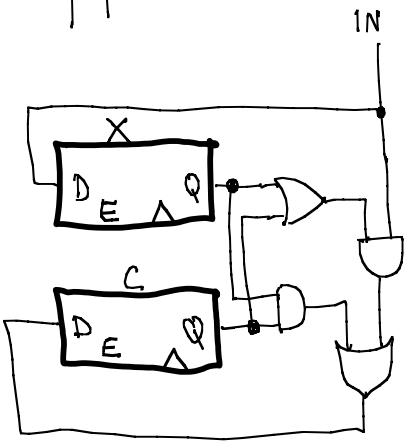


Carry function, w/ Karnaugh map + algebra

X.Q	IN	C.Q	C.D (majority)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

minterms ⇓ 8 AND + 3 OR



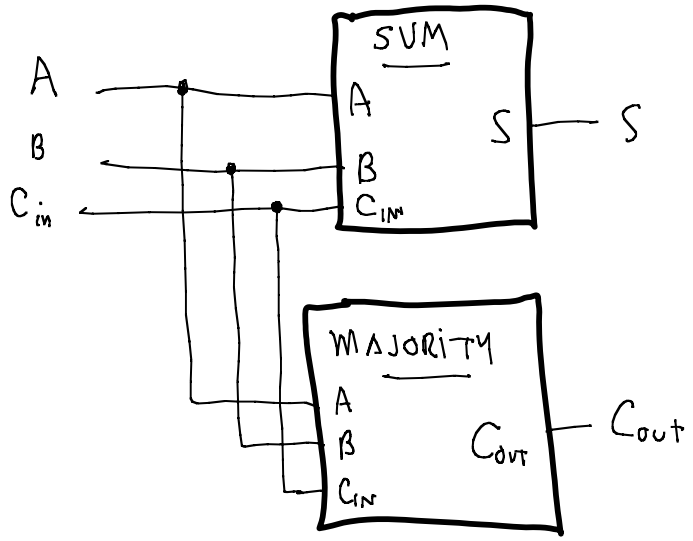


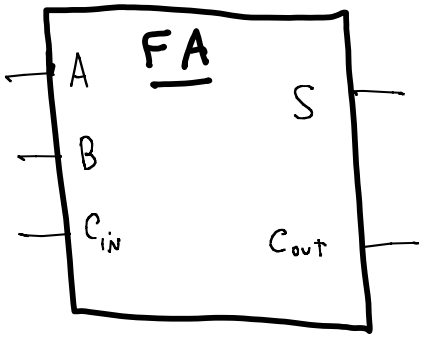
$$C.D = IN \cdot C.Q + IN \cdot X.Q + X.Q \cdot C.Q$$
$$= IN \cdot (C.Q + X.Q) + X.Q \cdot C.Q$$

[eliminates 3-input OR]

→ (Convert to NAND/NOR $\overline{Q}$ is available.)

Full Adder (FA)

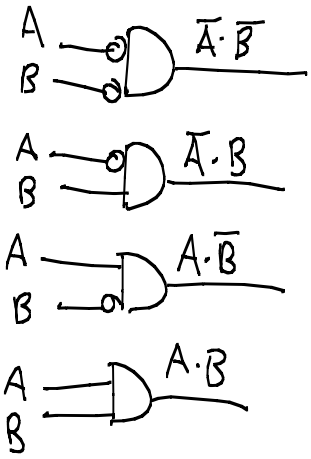


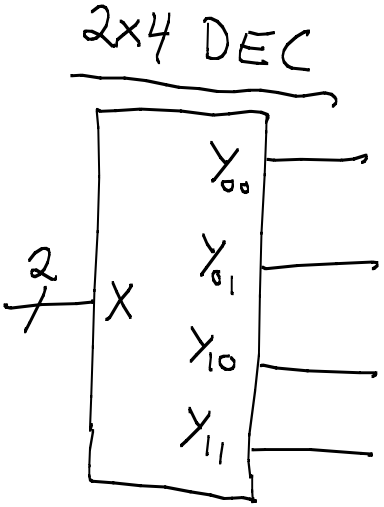


SMALLER FA circuit?

Can you find terms shared between MAJORITY and SUM?

PLA





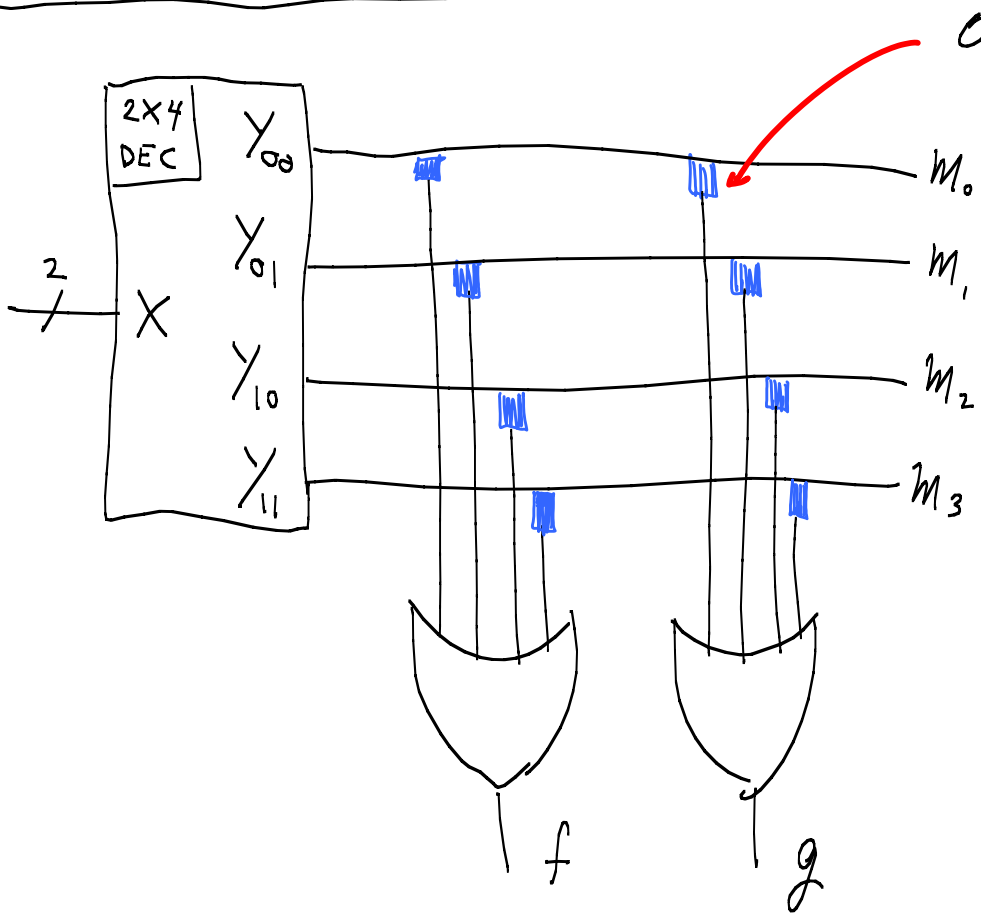
Probrammable Logic Array consists of two parts:

PLA, part 1.

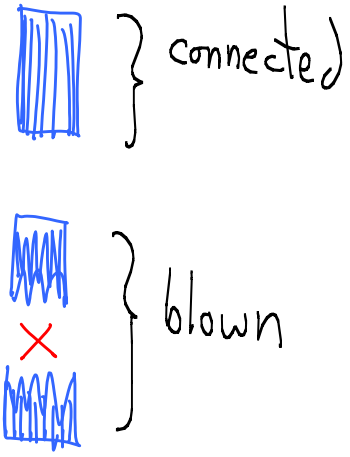
A decoder, which can be thought of in two ways:

a) a device that activates exactly on output depending on the code sent in.

b) a device that simultaneously generates all minterms for its input.



Optional Connection



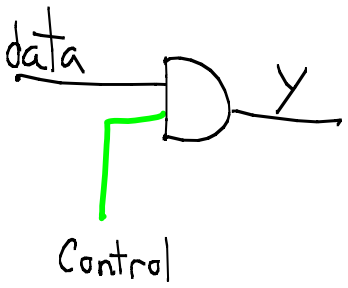
AND-OR ckt. for any f, g

PLA, Part 2.
A means of OR'ing minterms to produce function outputs.

Several ORs can share the same minterms: we can economically produce multiple functions at once.

The connections to minterm lines can be "blown" to disconnect them: this selects which minterms are included in the function.

MUX

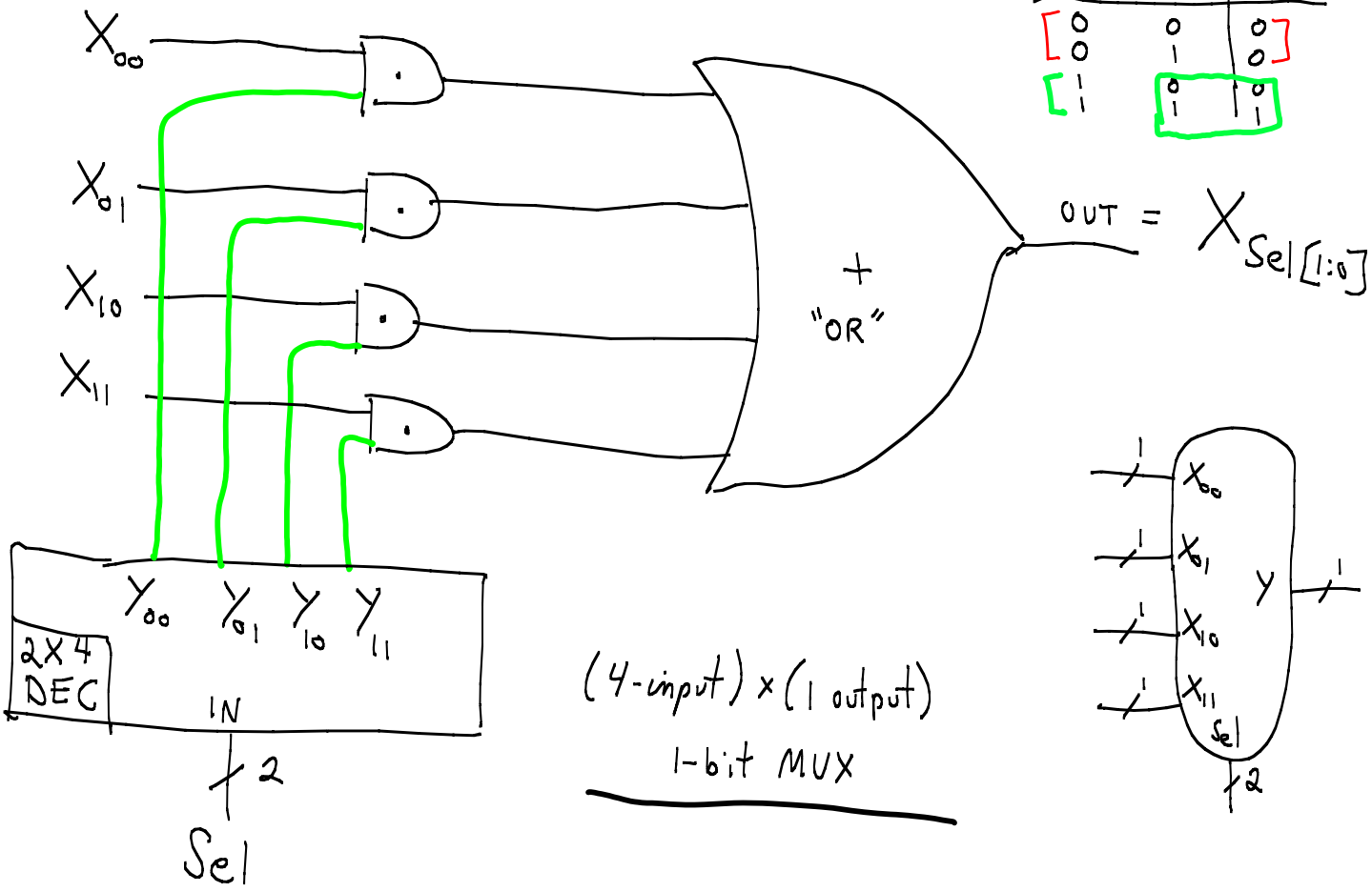


"gating"

Control = 0 : Y = 0

Control = 1 : Y = data

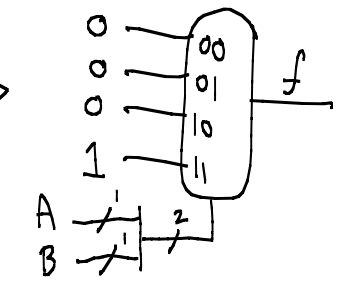
control	data	Y
0	0	0
0	1	0
1	0	0
1	1	1

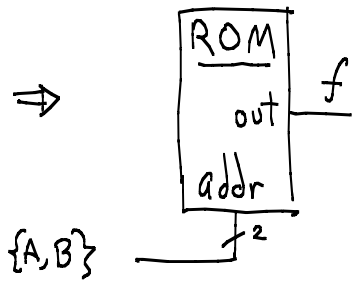


(4-input) x (1 output)

1-bit MUX

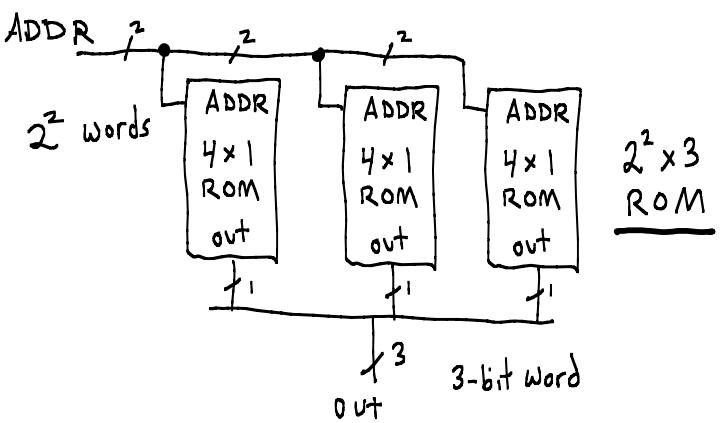
A	B	f
0	0	0
0	1	0
1	0	0
1	1	1



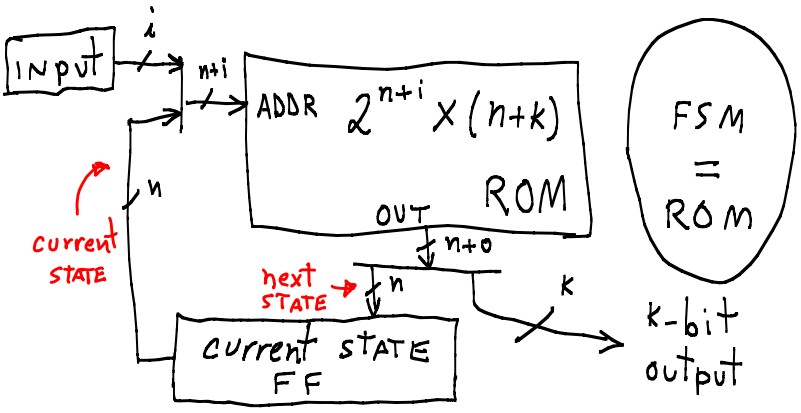


4 words,
1-bit word
=
4x1 ROM

Logic = ROM



2^2 x 3 ROM



FSM = ROM

FSM in ROM (n-bit state, i-bit input, k-bit FSM output)

(STATE, INPUT) is ROM address
n bits + i bits ===> 2^(n+i) ROM locations

(NEXT-STATE, FSM-OUTPUT) is ROM output
n bits + k bits ====> (n+k) bits per location

===> 2^(n+i) location by (n+k)-bit word ROM

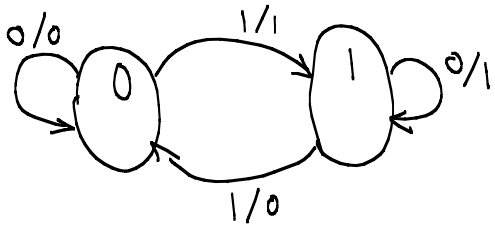
ANY FSM (Mealy or Moore) can be built as a ROM

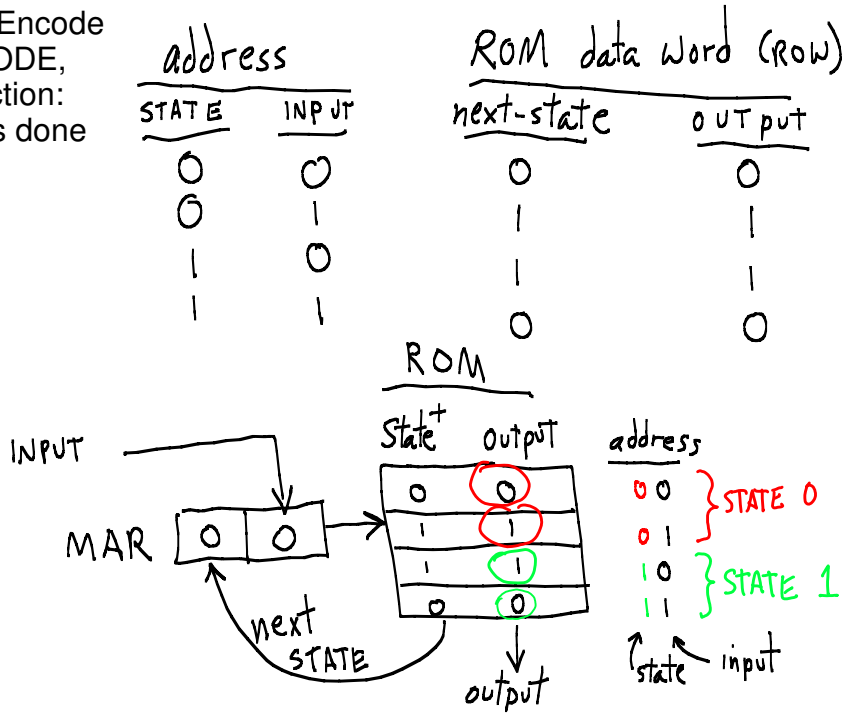
NOTE: A Moore machine's output depends only on state
===> use n-bit addresses, one ROM location per state.

BUT, next-state depends on current-state+input. Encode part of next-state function in ROM word as NS-CODE, and use external logic to calculate next-state function: next-state = f(INPUT, NS-CODE). This is what is done in the LC3's micro-coded controller.

Every possible FSM can be built as a ROM.

ROM is very large since there is a word for every possible {state, input} combination.





We can enumerate all ROMs (and consequently all TMs/digital-computers):

Concatenate ROM content from all words:

address	content
00	00
01	11
10	11
11	00

==> 01111000

List all n = i = k = 1 machines:
FSM-0, FSM-1, ..., FSM-256

List all n = i = k = 2 machines:
FSM-257, FSM-258, ...

and so on.

at clock tick:

-- { current state, current input } captured
-- output changes to match captured state/input

-- Every state row has same output
===> Moore Machine

-- Rows for state S have differing outputs
===> Mealy Machine.