LECTURE 5-6

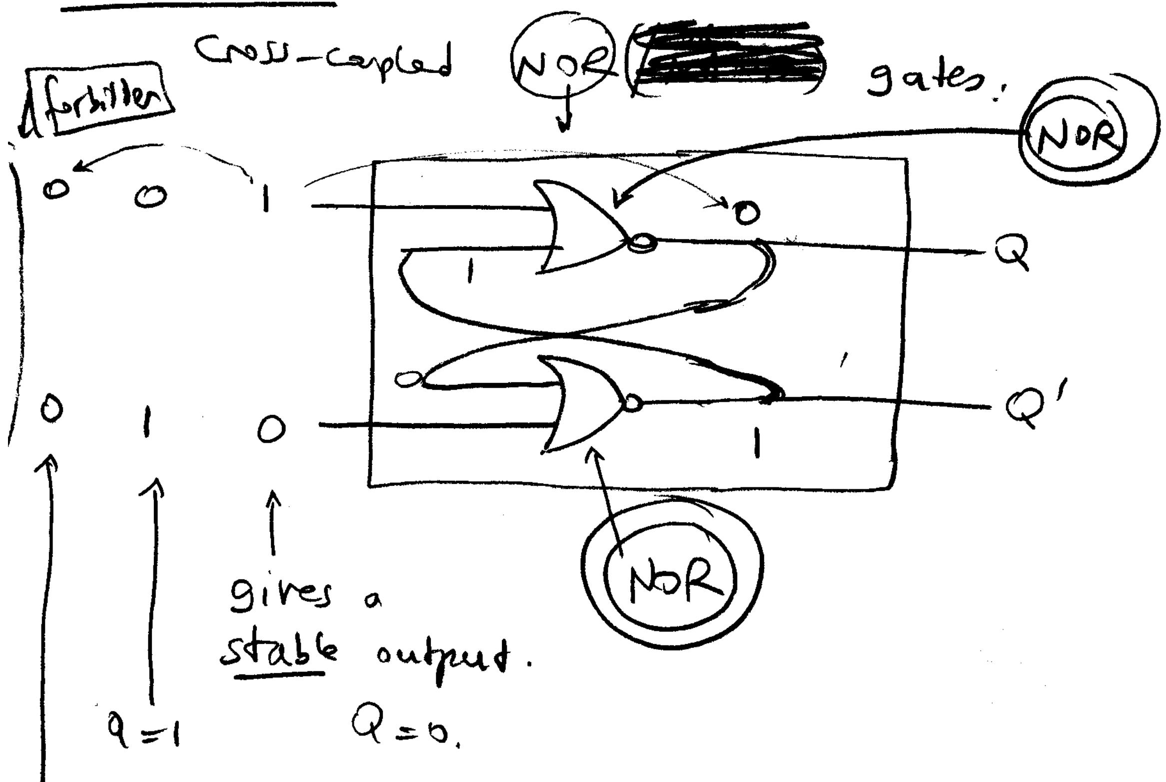
Sequential Circuits:

A circuit is combinational iff the output is determined completely by the current set of inputs. Otherwise, it is sequential.

- Sequential circuits are necessary to implement memory elements (i.e. elements that can remember their "state").

one of the simplest memory elements:

Basic Latch,



A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

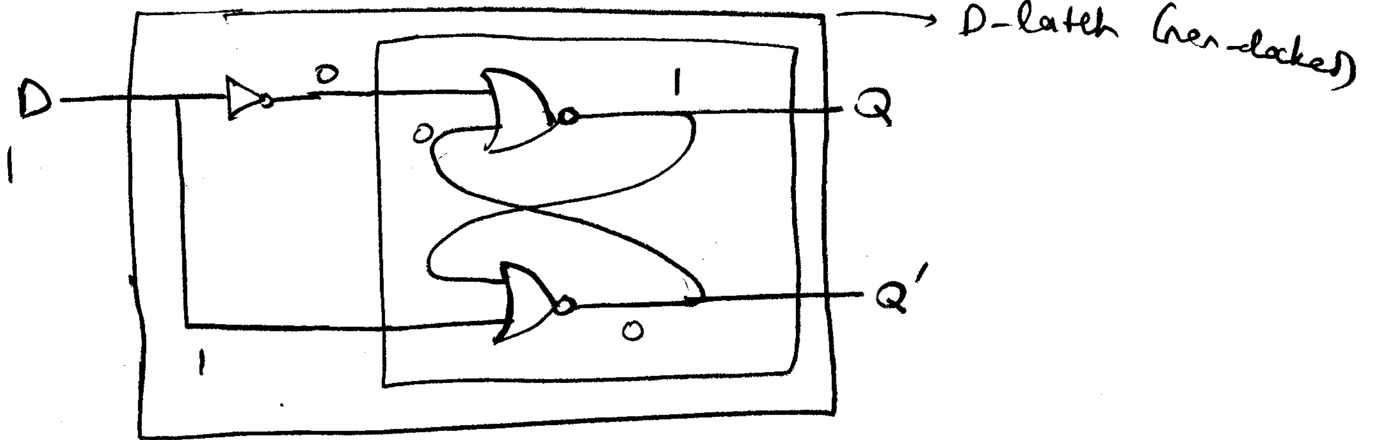
[0→1]

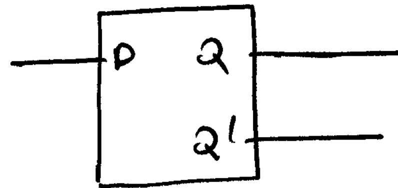
A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

[1→0]

holds the value it had originally

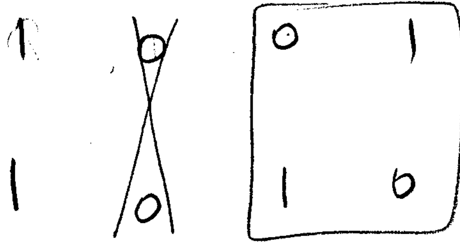
From this, we can build a (non-clocked) D latch as follows:



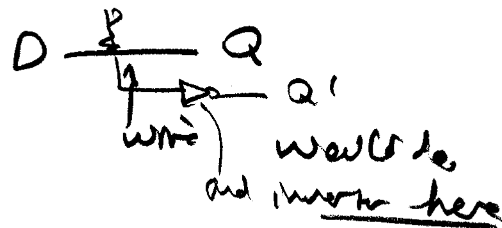


interface schematic.

Behavior:



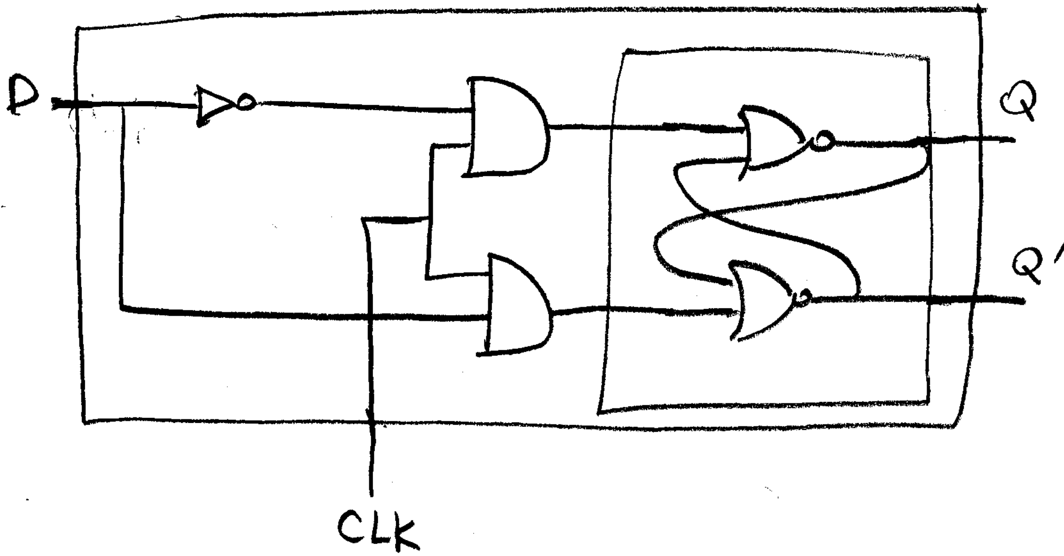
- Q gets D and Q holds the value of D
- Seems very simple & useless, because if this is all we wanted then



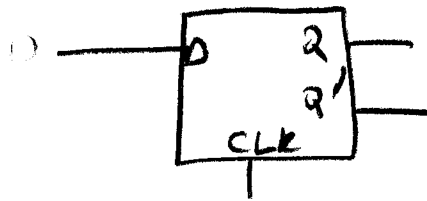
But: useful behavior obtained when clocked

(like an Enable)

Clocked D-latch:



Interface schematic:



So, when CLK == 1, Q=D ; and when CLK == 0, the basic latch hold the value of Q.

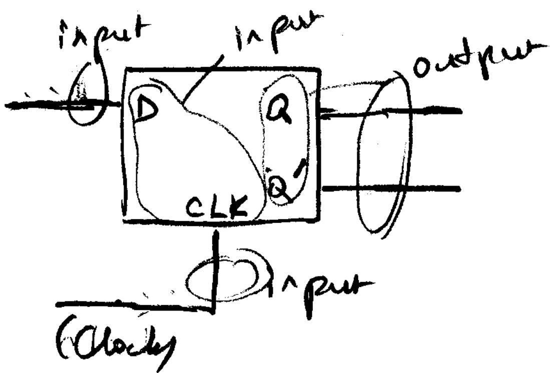
ECE 152A (Fall 2004)

[Lecture # 4]

[Reading: Ch 7.1, 7.3]

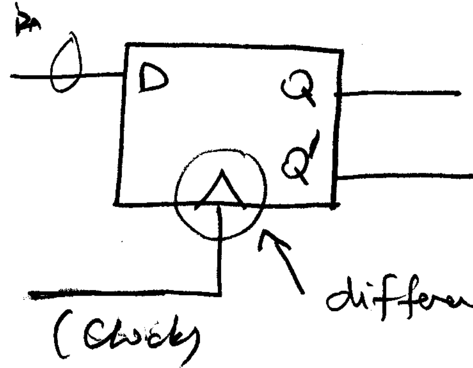
- Two key memory elements in synchronous design of sequential circuits:

- Clocked D-latch (clocked D latch)



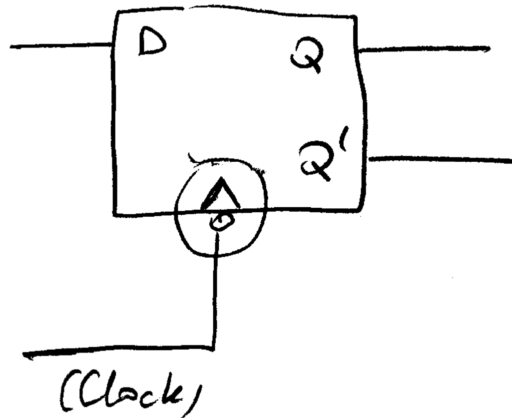
- D-Flip flop

(a) positive edge-triggered



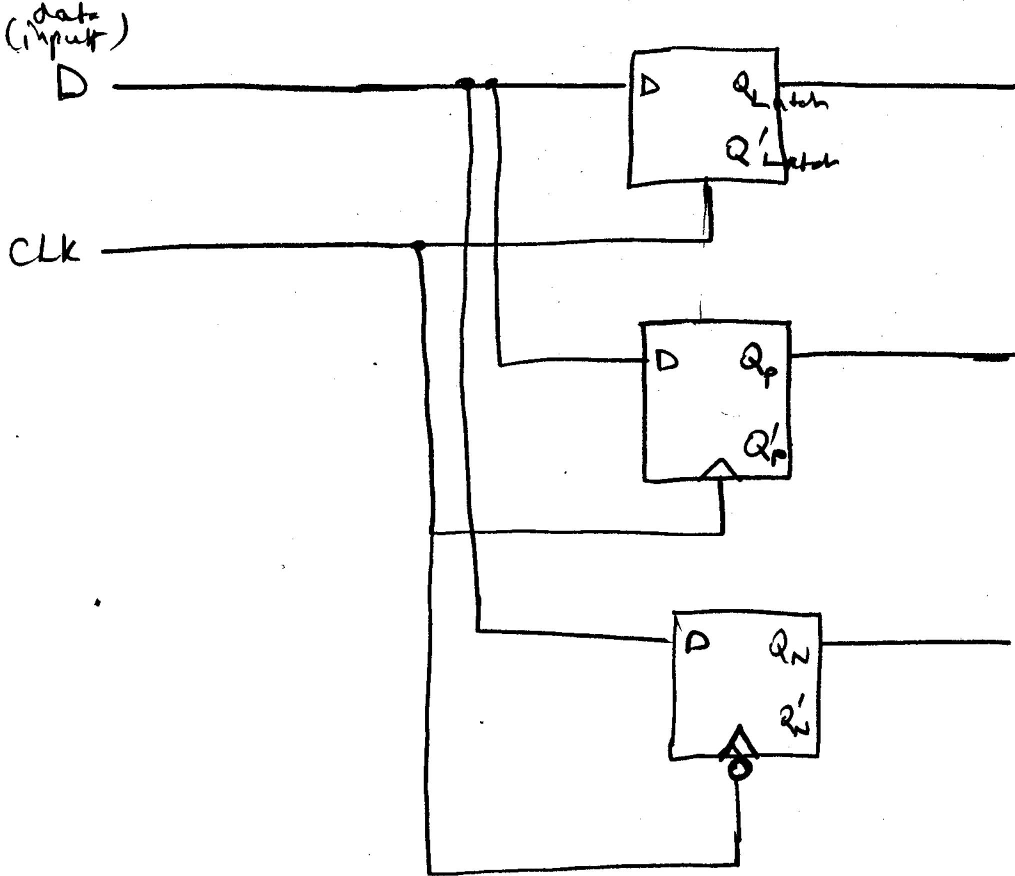
- Each of these can remember/hold 1 bit.)

(b) negative edge-triggered



– What is the behavior of these memory elements?

Let's connect a circuit this way for illustration,



Latch is transparent to its data input when the CLK is high — otherwise holds its last value

PET-FF samples its data input at the rising (positive) edge. — otherwise holds its last value

NET-FF samples its data input at the falling (negative) clock edge. ~~holds last value~~

~~timing diagram:~~

(Ideal operation)

CLK

t

D

t

initially assumed 0.

Q_L

t

Q_P

sample

hold until next rising edge

sample

hold

sample

t

Q_N

sample

sample

t

- "Active edge" of the clock :

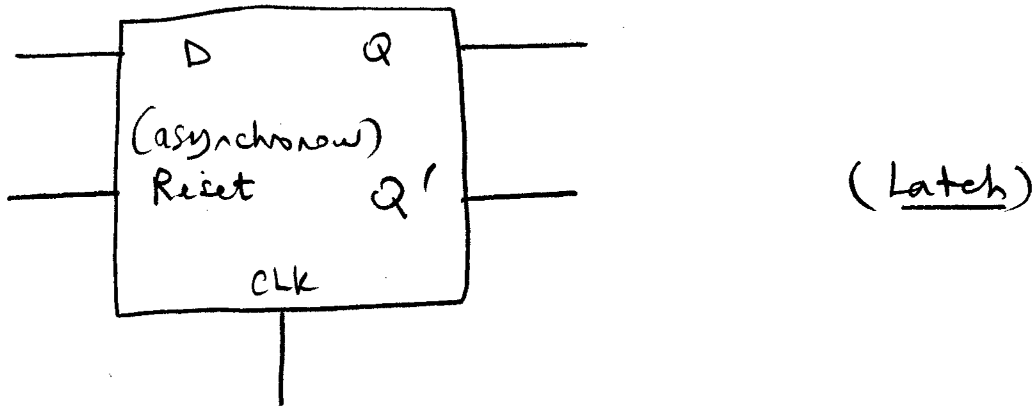
$$= \begin{cases} \text{rising edge} & \text{for a PET-FF} \\ \text{falling edge} & \text{--- NET-FF.} \end{cases}$$

- Ideal timing diagrams above since real latch/FF's show prop. delays & other effects that we will cover later

RESET DISCUSSION → see next page; then come back

– Above, we assumed that "somehow", the latches & ffps were initialized to zero (i.e. $Q_{initial} = 0$.)

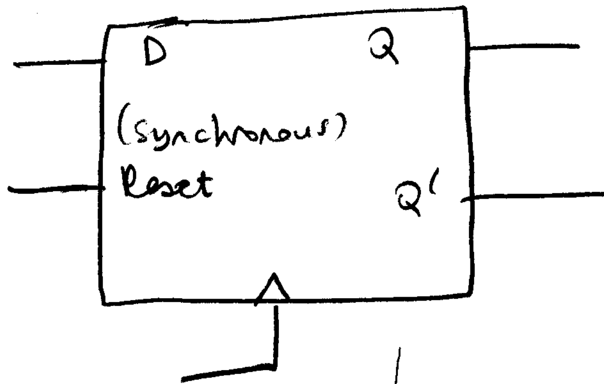
– You need an explicit <u>reset</u> input to achieve this:



(<u>Latch</u>)

If Reset == 1, then Q = 0 (no matter what the D input may be.). This kind of Reset is called <u>asynchronous reset</u> because the change in Q = 1 → Q = 0 can happen any time. ~~[This is a problem for latches — and we will see why (setup time requirement).] But for now, we will use this kind of reset.~~ ~~We don't like asynchronous reset because (as we will see later) can cause the circuit to fall into an unpredictable state.~~

Similarly, for ffp's:



Reset can be <u>asynchronous</u> or <u>synchronous</u>; we prefer <u>synchronous</u> reset (implemented inside flip-flop).

→ Reset on our ffps will always imply synchronous reset.

The ff's reset will be a <u>synchronous reset</u> (we will design it this way.). So: At the active clock edge, if the Reset == 1, then Q = 0. Otherwise, Q = D.

– This is the behavior of the FF with synchronous reset.

Ex1 <u>Timing diagram:</u>

Assume PET-FF

CLK

Reset

D

Q_{PET-FF}

undefined

initialized to <u>zero</u>

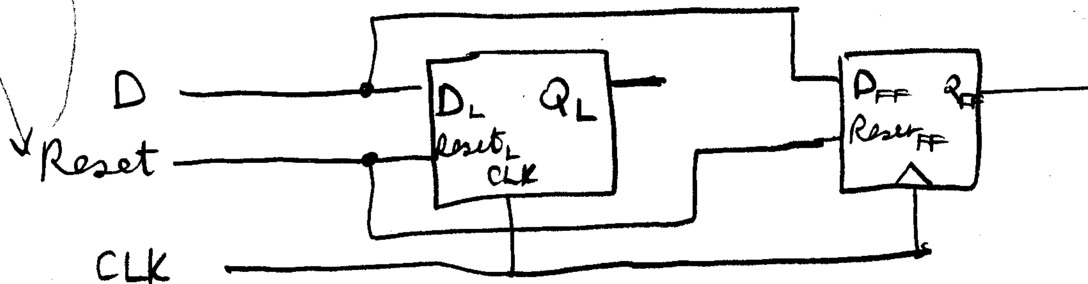
Q_{Latch}

same value as D

holds last value when clock ↓.

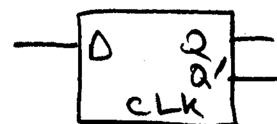
since reset == 0 and D == 1 & CLK high

Stays zero! Since Reset = 0.

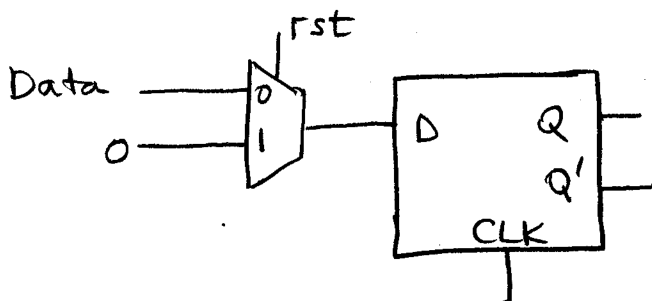


NEXT-STATE EQUATION:

① Clocked D latch without Reset input:



[Can achieve reset as follows: (do later)



+ will be reset only when clock goes high.
− rst must stay high until clk goes low.]

— "Next-state eq'n" for a clocked latch (without asynchronous Reset) shows the next value of Q after any change in inputs assuming CLK == 1.
(does not apply when CLK == 0; we know that it holds the value then.)

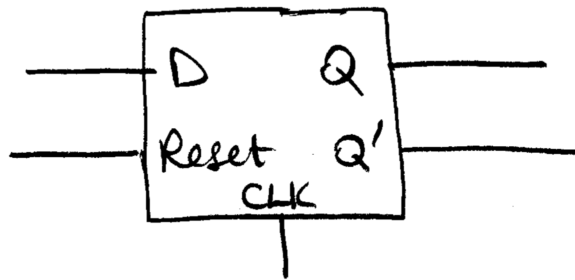
D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

"state table"

(Copies D to Q no matter what previous value of Q is.)
["memory" aspect comes in when CLK == 0.]

Next-state equation: $Q^+ = D$ (clocked D latch without Reset.)

② Clocked D latch with asynchronous Reset :



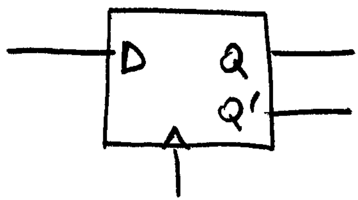
- Next-state eq'n for a clocked D latch with asynchronous Reset must include the CLK input in state table (since we can no longer assume that it "holds" in CLK == 0 : Reset may happen when CLK == 0)

Reset	CLK	D	Q	Q^+
1	X	X	X	0
0	0	0	0	0
	0	0	1	1
	0	1	0	0
	0	1	1	1
	1	0	0	0
	1	0	1	0
	1	1	0	1
	1	1	1	1

Rows with CLK = 0: "hold Q".

Rows with CLK = 1: "copy D"

Next-state eq'n: $Q^+ = (Reset)'[CLK \cdot D + CLK' \cdot Q]$



③ D flip-flop without Reset:

- Next-state eq'n for a D flip-flop shows the next value of Q right after the active clock edge.

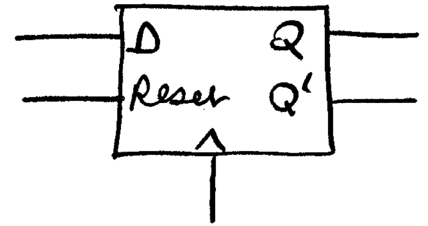
- State table:

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

- Next-state eq'n: $\boxed{Q^+ = D}$

* same equation as for a clocked D latch without Reset, but very different meaning.
  - Latch: Q^+ occurs after any change in inputs.
  - FF: Q^+ " at the active clock edge.

④ D flip-flop with (synchronous) Reset :



- NSE shows the next value of Q right after the active clock edge.

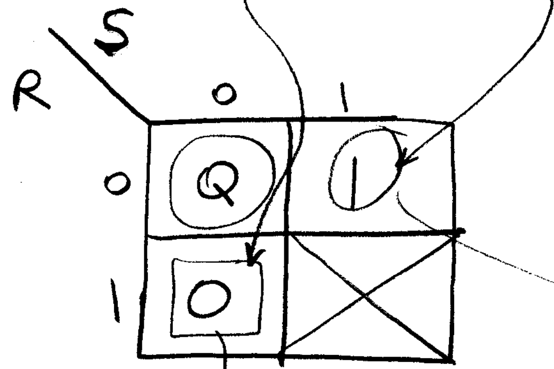
- State table:

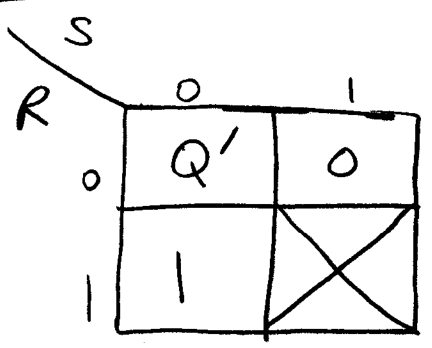
Reset	D	Q	Q^+
1	X	X	0
	0	0	0
	0	1	0
	1	0	1
	1	1	1

- Next-state equation: $Q^+ = (Reset)' \cdot D$

Continue here

R-S latch:





$$R = Reset + CLK \cdot D'$$

$$S = (Reset + (CLK \cdot D)')'$$

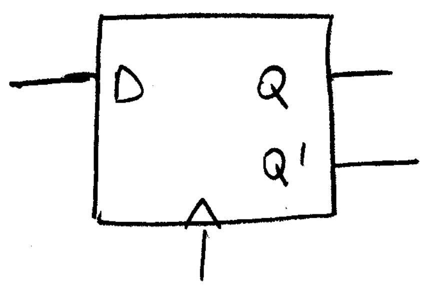
$$= Reset' \cdot (CLK \cdot D).$$

i.e. implement each ON-term by S and each OFF-term by R.

③. Implement a D flip-flop without Reset terminal,

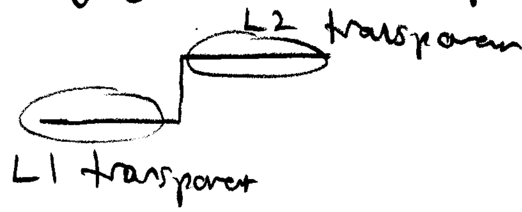
How do we implement a flip-flop?

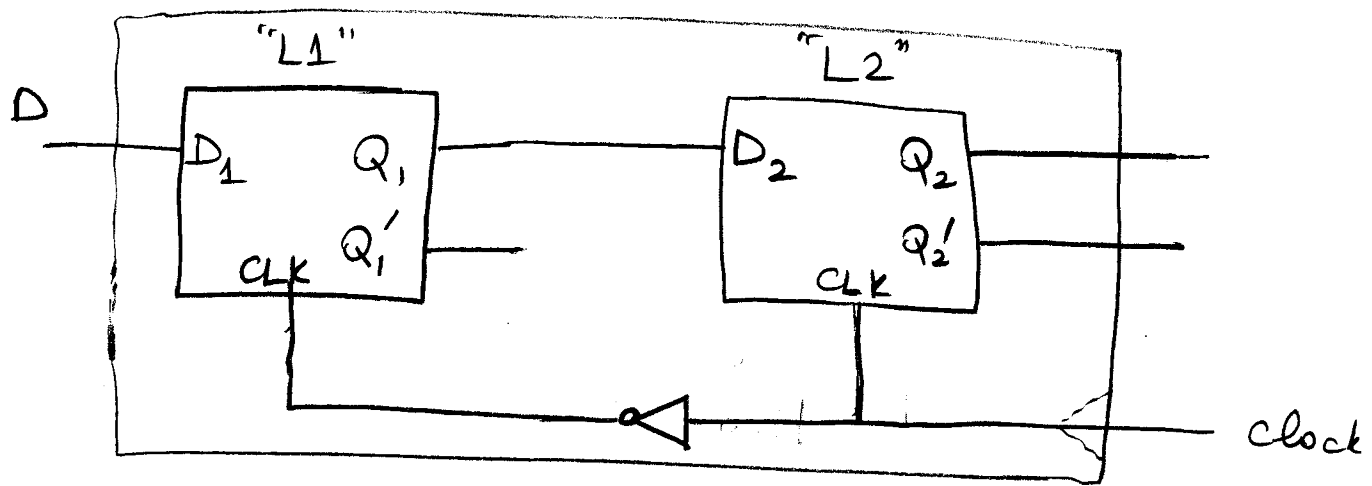
- Here, implement it from 2 latches in cascade.



Main idea: Have the first latch be transparent when the CLK is low. Have the second latch be transparent when

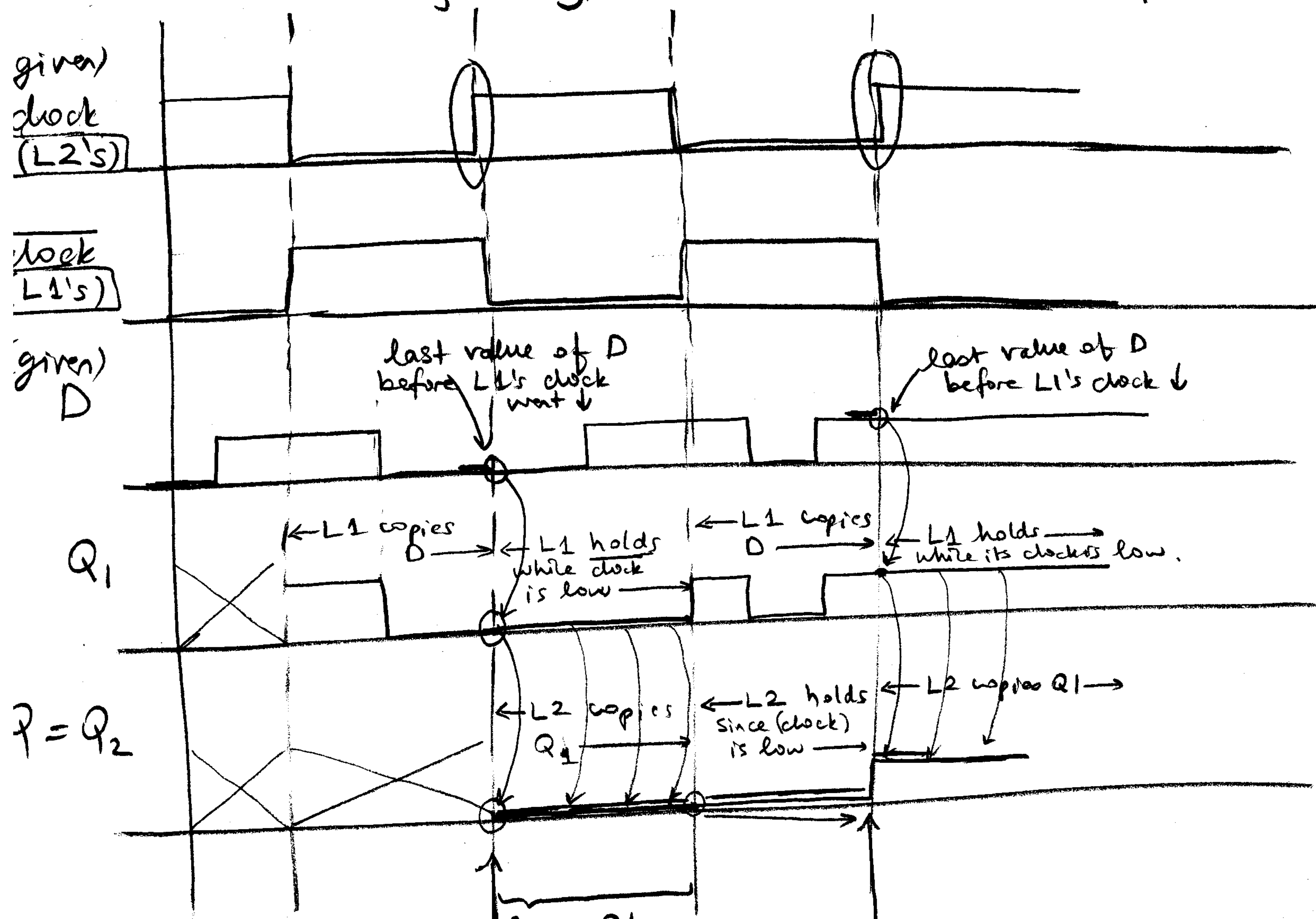
the CLK is high. This way you get "sampling on the positive CLK edge."





This is called a **master-slave** (refers to the implementation) D flip-flop. The first latch (L1) is considered the master and the second latch (L2) is the slave since L2 must follow what L1 (its master) sends it.

- Draw the timing diagram to check its behavior:



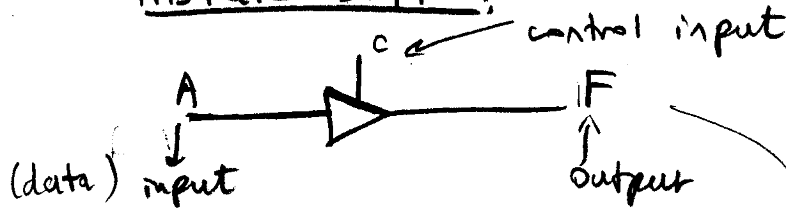
Effect: D's value of 0 is sampled at rising clock edge, (and held until next rising clock edge)

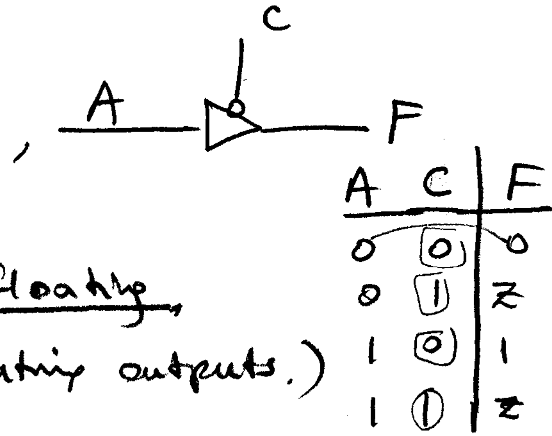
D's value of 1 is sampled at rising clock edge (and held until the next rising clock edge).

EXERCISE: Build a ... out of 2 latches

— Now, we introduce a very useful combinational block!

tristate buffer:





A	C	F
0	0	0
0	1	Z
1	0	1
1	1	Z

If $C==1$, $F=A$; otherwise B is **floating**.
(So far, we had not allowed any floating outputs.)

So:

A	C	F
0	0	Z
0	1	0
1	0	Z
1	1	1

Z ← (floating; high impedance)

If we used combinational logic blocks to implement this,

A	C	F
0	0	X
0	1	0
1	0	X
1	1	1

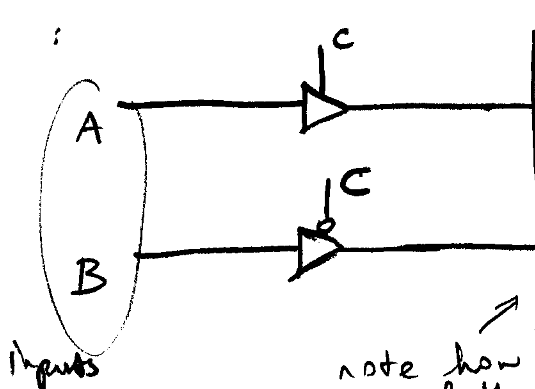
X ← don't care

$F = C \cdot A$ (min SOP form)

(produces 0 when $C==0$)

But tristate buffers can be used in ways that well-defined combinational cannot.

Ex



F : output

$F = A \cdot C + B \cdot C'$
(implements a 2:1 mux)

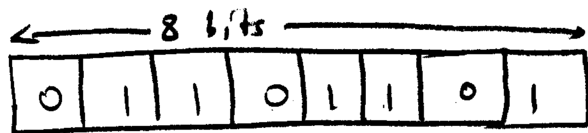
→ requires much less hardware than the AND-OR gate implementation.

note how F is left floating at each end (cut off)

In computer architecture, we need to hold >1 bit.

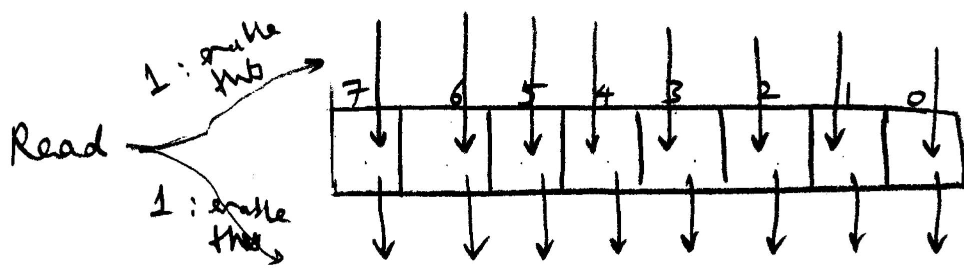
e.g. Let's say I want to perform operations on <u>bytes</u>.

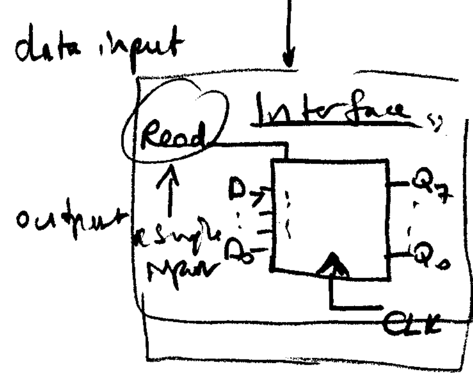
1 byte = 8 bits. You need to hold those 8 bits together somewhere, perform an operation and perhaps write the resulting byte somewhere else.



] a "register" holds this.

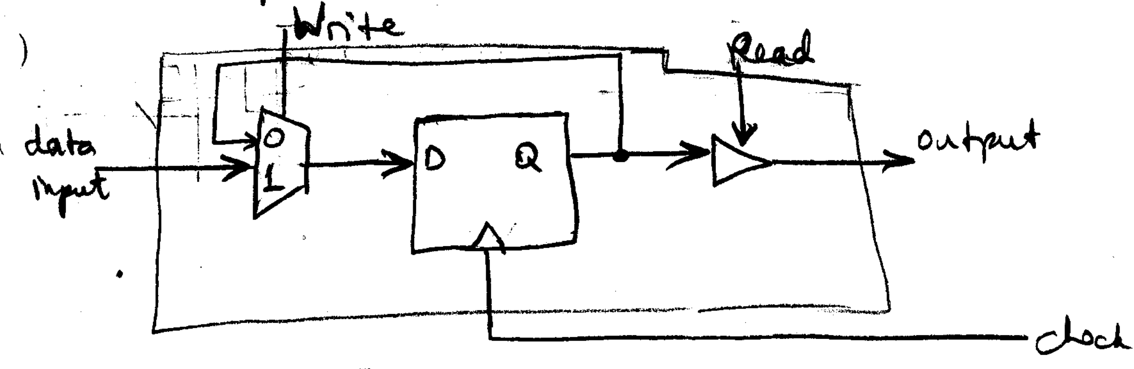
<u>Ex)</u> Design an 8-bit register using flip-flops. The circuit will have ~~the following inputs:~~ a 1-bit control input Read and 1-bit control input Write. When Read == 1, we read the register contents to output. When Write == 1, we write the register with new content.





[Ask class;] how to do this.

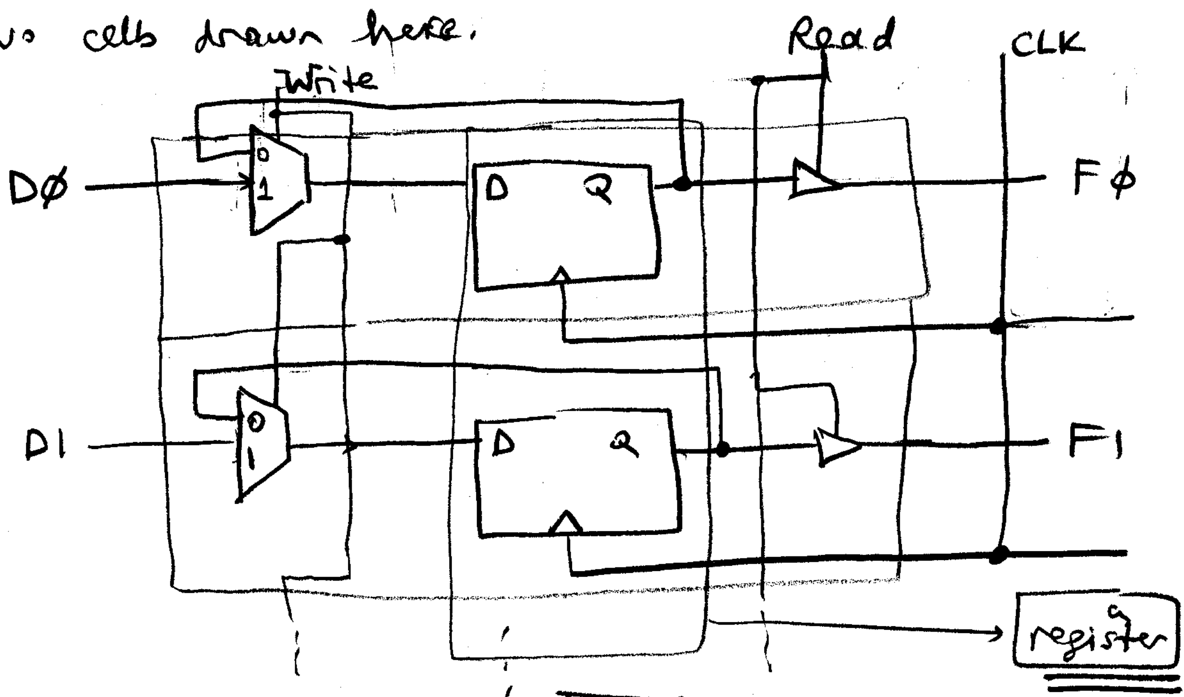
Since all operations will be identical on each bit, let's focus on 1 bit only



• control circuitry must make sure that Read and Write not asserted at same time.

Replicate this cell 8 times to get the design

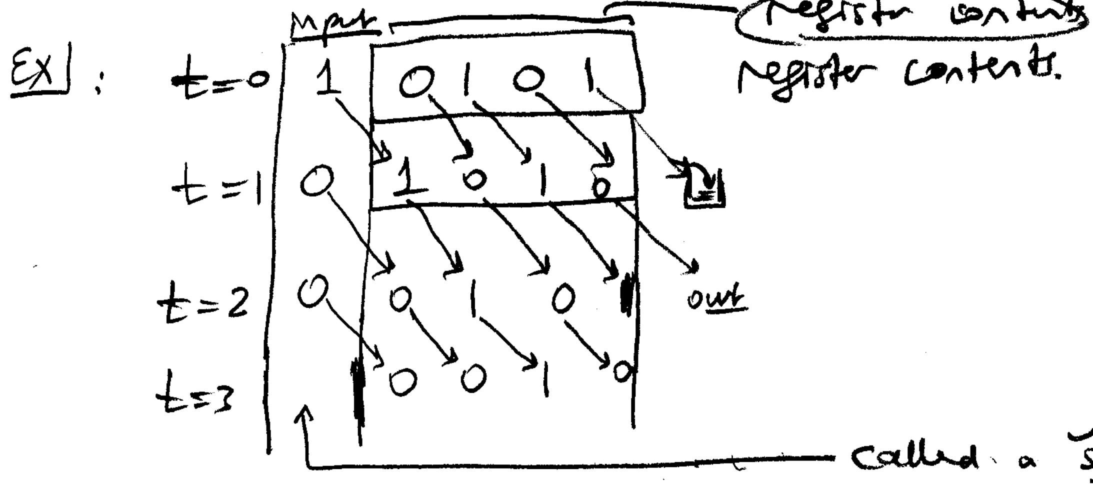
Two cells drawn here.



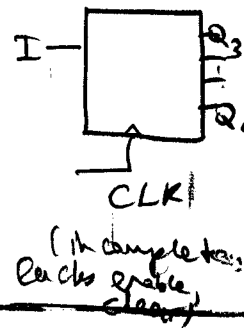
The control lines Read, Read run vertically.
The data lines run horizontally
Note how the control signals need to be distributed to the cells

Now in computer architecture, we also want structures that can achieve the following:

Ex:



This is called a shift-register. → Interface (Schematic): (4-bit s.r.)

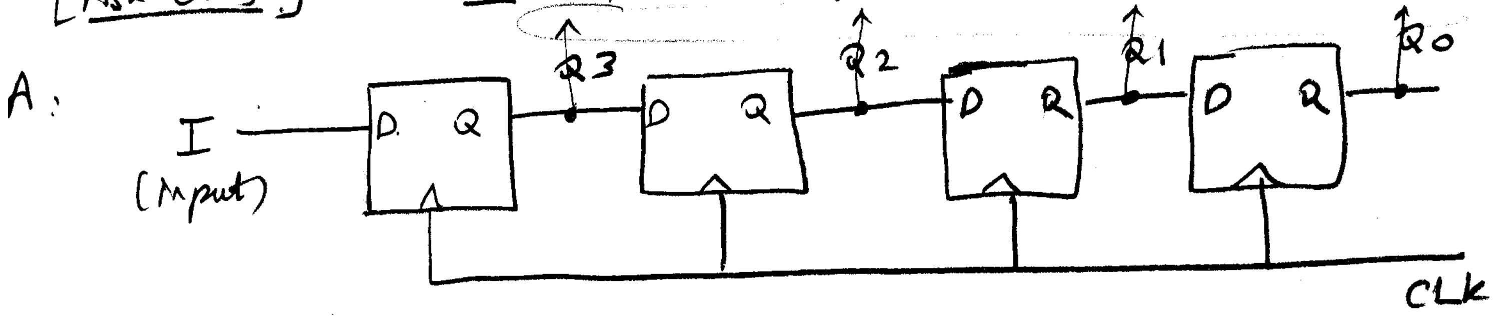


[Example usage: Design a "divide by 2" circuit.
— divide by 2 means, you need to shift the contents to the right by 1. (we will talk more about arithmetic circuits later.)]

Q: How do we implement a "shift register" using ffps?

[Ask class]   Ex: 4-bit shift register

A:



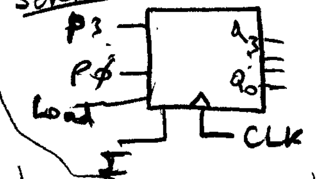
At each positive edge, contents will shift to the right by 1 (and Q0 will get overwritten)

Now, what if we wanted:

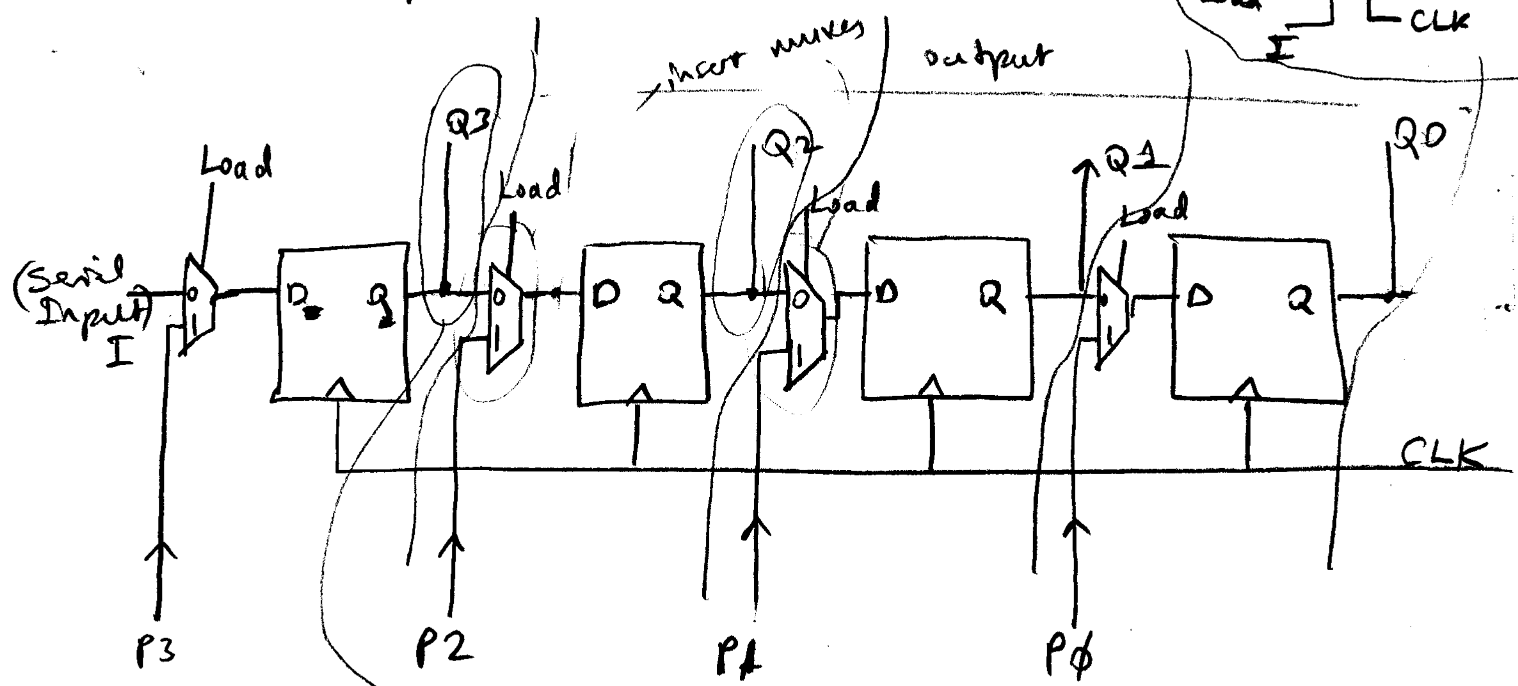
* Input: . Load : ==1 : Load 4 bits into the shift register

. Load : ==0 : shift using a serial input — and we want

(called a Parallel-Access Shift Register)

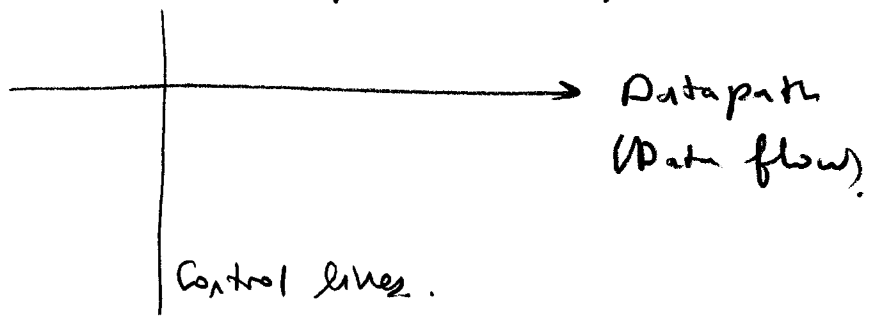


— How can we implement this?



make sure to take the output from the output of the ffps — that way they will be Synchronous. Do not put them after MUX!

Note the structure:



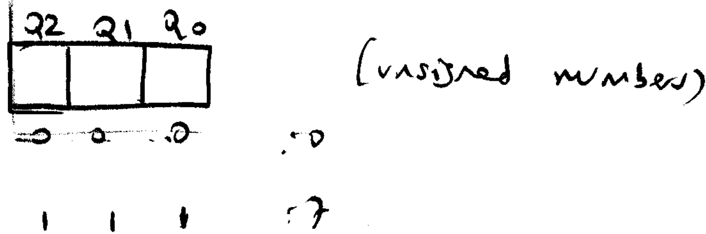
- Very often, we need <u>counters</u> in digital circuits.

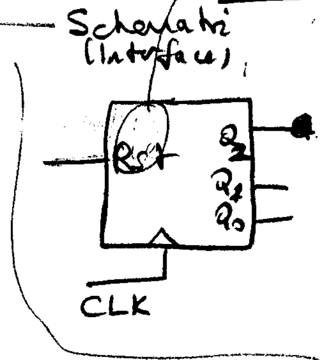
<u>Example</u>: Count the number of cars on street.
If #cars ≥ 1000, raise a flag.

- Counters can be implemented using flip flops

<u>Example</u>: Design a circuit that counts the number of clock pulses (rising clock edges), from 0, 1, 2, ..., 7 and then wraps back to 0 ... 7 and again, and Resets to 0 when Rst is asserted. (synchronous)

<u>Answer</u>: First, find a representation; need 3 <u>bits</u>





Then, let's display the operation, will use a ff for each.

clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

time ↓

<u>Key question is</u>, how can we decide what the next state of each flip flop will be
- We need to write each next state in terms of the previous states such that it will apply to every transition

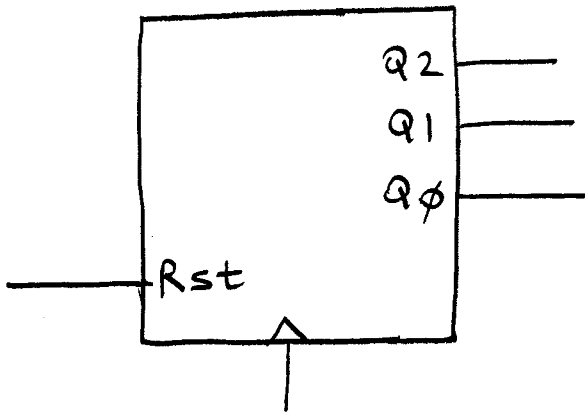
~~[Another counter design using more formal/general method:
- illustrate how to implement load, enable, clear, reset, for counters.]~~

~~Let us return to our examples~~

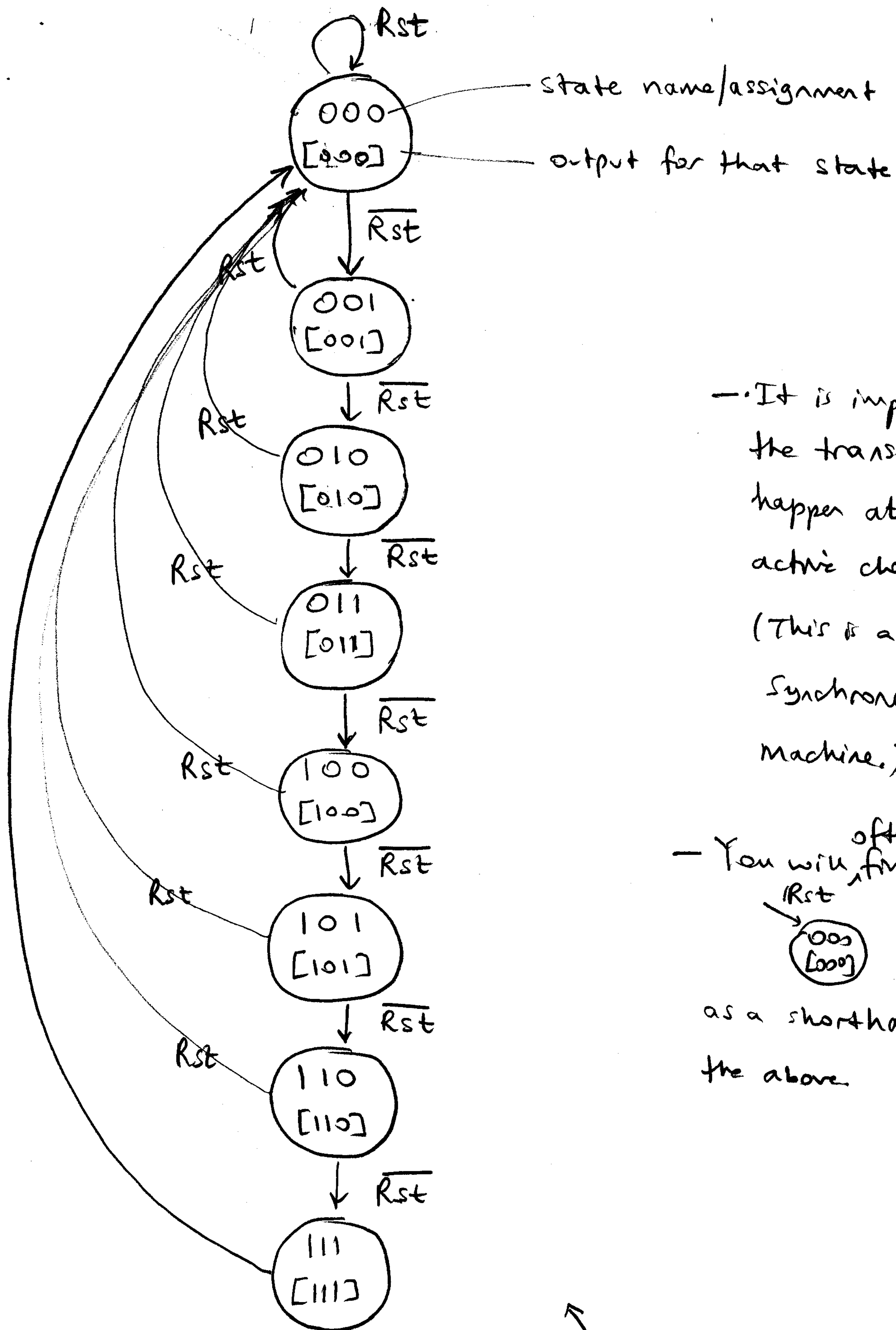
- counting sequence: 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, ...
  (000) (001) . . . (111)
  (counts the # of active clock edges)

- Each count represents a "state".
- When counter is <u>reset</u>, it should go to 0, in a <u>synchronous</u> fashion
- See next page for the state diagram.

- Interface schematic:



- Rst is the only input here besides the clock.



– It is implied that the transitions happen at the active clock edge. (This is a synchronous machine.)

– You will often find that

as a shorthand for the above.

↖ called a <u>state diagram</u>.

State table:

(dependence: current state → output)

Input	current state			next state			output		
Rst	Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	Z_2	Z_1	Z_0
1	q_2	q_1	q_0	0	0	0	q_2	q_1	q_0
0	0	0	0	0	0	1	0	0	0
	0	0	1	0	1	0	0	0	1
	0	1	0	0	1	1	0	1	0
	0	1	1	1	0	0	0	1	1
	1	0	0	1	0	1	1	0	0
	1	0	1	1	1	0	1	0	1
	1	1	0	1	1	1	1	1	0
	1	1	1	0	0	0	1	1	1

(placeholders)

~~[A finite state machine in which the output is a function of only the current state (and not of the current input) is called a Moore machine.]~~

(later)..

Mapping the state table to a D flip-flop implementation (by hand):

- Do a K-map for each next-state variable and each output variable. Then minimize logic.

— In this example:

outputs:

$$Z_2 = Q_2$$
$$Z_1 = Q_1$$
$$Z_0 = Q_0$$

} already in minimum form

— Examine next-state variables:

$$Q_0^+ = (Rst)' \cdot (Q_0')$$

read directly from state table.

Q_1^+ (for $Rst == 0$) (Rst==0)

Q_0 \ Q_2, Q_1	00	01	11	10
0	0	1	1	0
1	1	0	0	1

$$Q_1^+ \Big|_{Rst==0} = Q_1' \cdot Q_0 + Q_1 \cdot Q_0' \quad (= Q_1 \oplus Q_0)$$

$$Q_1^+ = (Rst)' \cdot [Q_1 \oplus Q_0]$$