# SEQUENTIAL LOGIC

Till now, we have only looked at Combinatorial Circuits

→ Combinatorial Circuits require present input & give present out

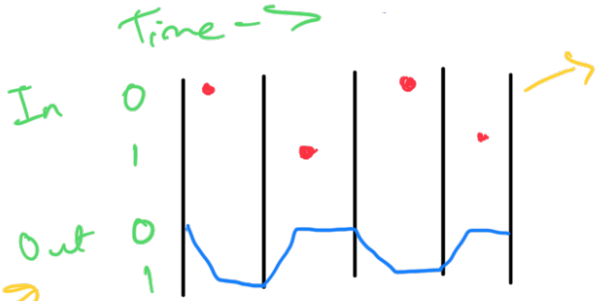
→ Sequential Circ use present inp and past input to give present output.

Lets take the Example of ALUs. ALU required a lot of chips but for a particular input all the operations were performed with that inputs, previous x or y values were not used.

↳ This is an example of Combinational Circuits.

Time is Continous in real World but in Computer Systems it can be hard to understand what's going on in continous time bcoz Continous time can have multiple inputs & outputs at varying time.

Real World Ex → Lets say we have a not gate & it take same time to function. Time is moving forward




→ At every sec we send an input to not gate, bad dot represent input at every second.

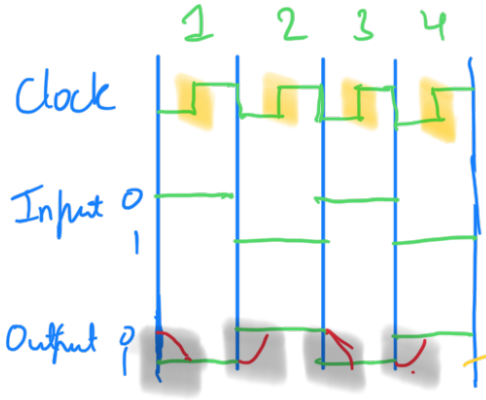
In this example we take time to be 1 sec but it can be anything

→ In real world input take time to switch and during that time they spit out wierd outputs so we use discreate time

## Discrete Time

We set up a Clock. This Clock ticks at a predefined time and we only perform actions at that point in time.

↳ Ex: Again we have a not gate & Let look at it with a clock



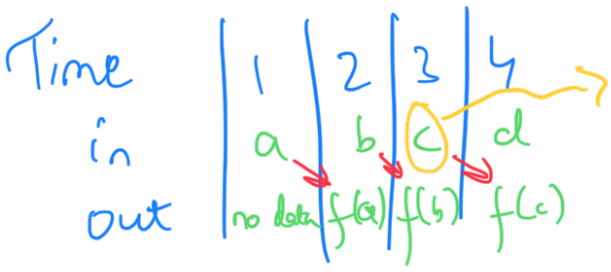

Throughout the time input stays same

→ Time is discreate. We don't care about 1.111 or 2.12 etc. we only care about time at 1, 2, 3 ....

Red lines indicate the real world. but we grey that portion out, ignore it & only care about green Vals.

# SEQUENTIAL CIRCUIT

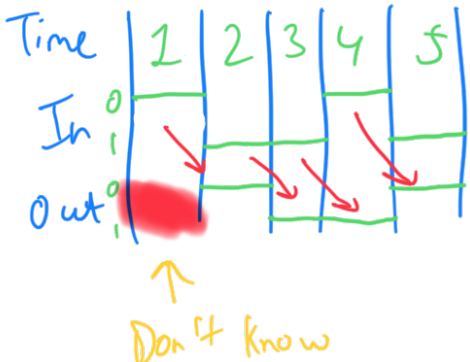
Here we need to take the input from previous time step So unlike our previous function $out(t) = f(in[t])$, here it will be $out[t] = f(in[t-1])$




(basically is f(a) & D is f(f(f(a)))

DATA FLIP FLOP

$out = in[t-1]$

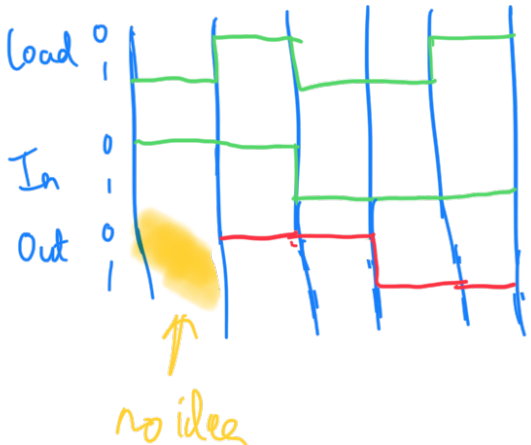


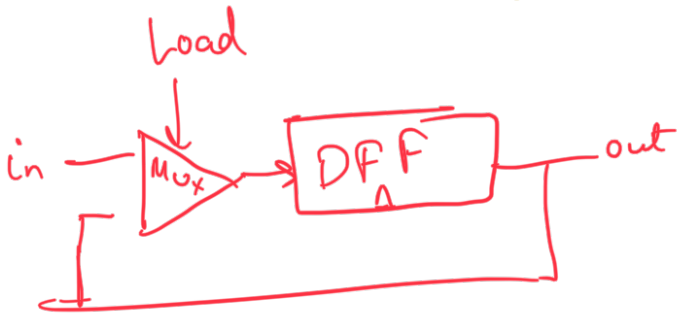

# 1 BIT REGISTER

It remembers a state or bit untill we tell it to remember a new Val.


if load(t-1) = 1 then out = in(t-1)
else out = out(t-1)

↳ Previous output thus we remember it.




DFF stands for Data Flip Flop. It is a chip given to us. So, lets make a 1 Bit Register

Remember that a clock ticks & tocks, so in our program tock state & register changes

output is changed ... in the tick state

Ex -> We want to store value.

Tick { in = 100, load = 1 } Register = 100, out = 0

Tock { in = 100, load = 1 } Register = 100, out = 100

PROGRAM

-> Mux (a = Prevout, b = in, sel = load, out = tempout);

DFF (in = tempout, out = out, out = prevout);

Here we made 2 out pins bcoz we can't use "out" again

Remember that every thing is run in one "swoop"!

# 16 BIT REGISTER

This is quite straight forward. We just need 16 1-Bit registers.

In = In[16], load

out = Out[16]

Bit (in = in[0], load = load, out = out[0])

Bit (in = in[1], load = load, out = out[1])

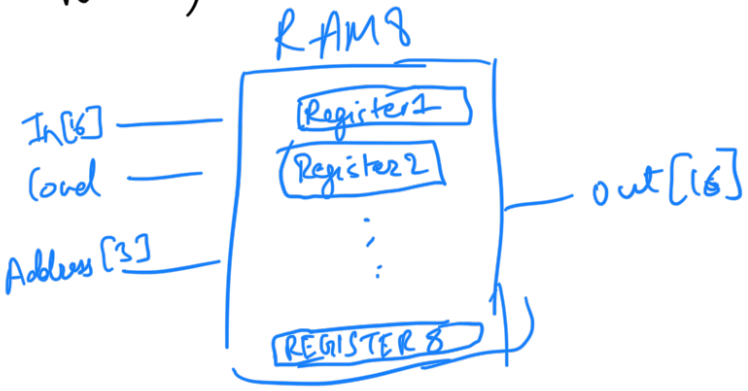
:

t...t[15])

Bit [in=in[15], load=load, out=out[15]]

# RAM 8

RAM consists of several 16 bit registers. Our goal is to read or write any one of these register. Unless we want to edit these values, the RAM must store them forever.




The address is used to access 1 of 8 registers, load tells us whether to change it or not & Input is our good old input.

address

0 0 0 -> Register 1
0 0 1 -> Register 2
1 1 0 -> Register 7
1 1 1 -> Register 8.

We need this functionality

We need a way to choose one of these 8 Registers. So we need something like this ->




Ring A Bell?

DMUX8WAY!!!

DMUX 8Way(in=load, sel= address, a=a, b=b ... h=h);

This will make one of the 8 out as 1 and others 0. We will use this as load for our Registers.

Register(in= in, load=a, out= R1);
Register(in= in, load=b, out =R2);
⋮
Register(in= in, load=h, out= R8);

Now we need to find a way to choose b/w these 8 Registers!

MUX8Way16(a= R1, b=R2, ... , h= R8, sel= address, out=out);

You can create RAM64, RAM512 etc, in the same way. So we won't discuss RAM further!

# PROGRAM COUNTER (PC)

In -> in[16], load, inc, reset
Out -> out[16]

| load | inc | RST | Out |
|---|---|---|---|
| | | | in |

Reset overrides

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | in+1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | new in |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | new in |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | |

everything 0 > 0
1 out is 0.
Next is load. If its 1, out is in.

This was the toughest chip for me so far. I tried a lot of times & for some reason increment never worked. But then eventually I got it!

If we look at our if conditions, reset overrides all so ....

The code below is written in a way to aid understanding. the number before the code is the actual line number.

4 Mux16(a= loadOut, b=false, sel=reset, out= tempOut);

↑ We will get this from our load Output which is the second in our if conditions.

3 Mux16(a=incout, b=in, sel=load, out= loadOut);

↑ We will get this from our incrementor

2 Mux16(a= register value, b=incremented, sel= inc, out= incout);

↑ value stored in the register

We will get this from inc16

Mux16(a= register value, out= incremented);

(1) Inc16(in= ...

At this point, we have our tempOut from the Mux16(.... sel=reset). But there is a problem. We haven't specified our register yet & there is no output-

Also, in the case load is 0 & inc is 1 & reset is 0, we want to change the value in register, which currently does not happen.

If you look at the truth table above, you will see that we want to change the value in register when reset or inc is 1 even when load is 0! To do that ↓

Or(a=inc, b=reset, out=newload);
Mux(a=newload, b=load, sel=load, out=finalload);

if load is 0, we want to use new load because we might have used reset or inc but when load is 1, we may want to store the input value, this will also work when reset is 1!

Finally,

Register(in=tempOut, load=finalload, out=out, out=registervalue);