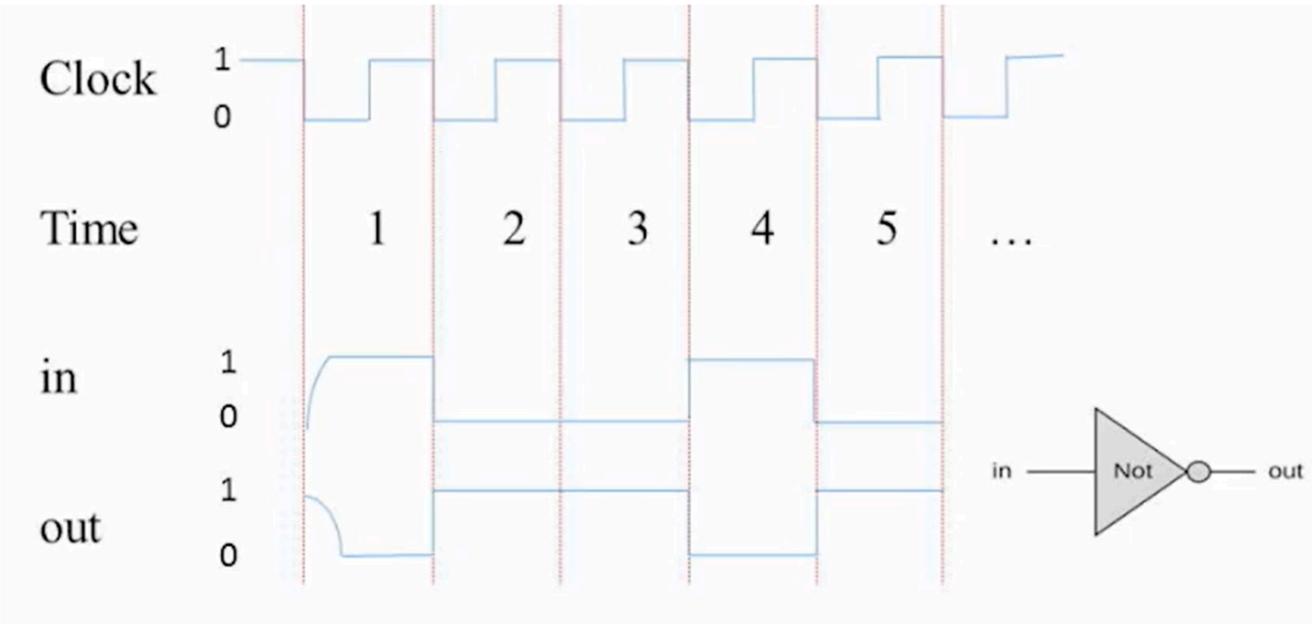
Module 3 - Sequential Logic and Memory

Combinatorial Logic and Sequential Logic

- Inputs are fixed and unchanging
- Output is just a function of input
- Output is computed instantaneously

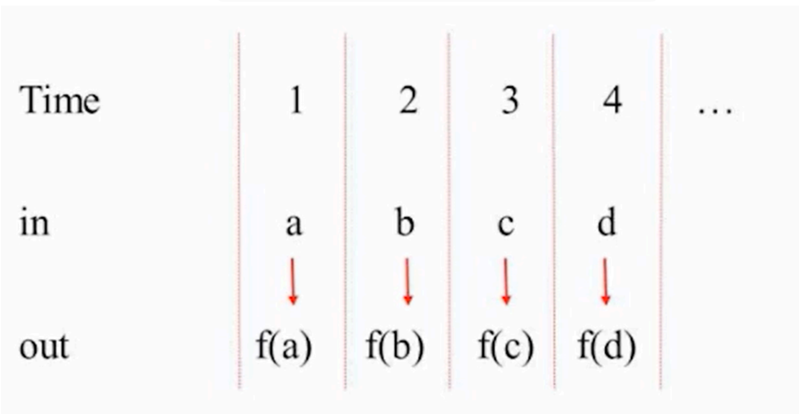
In computers, instead of working with continuous time, we work with discrete/integer time to let the states in the computer stabilises.



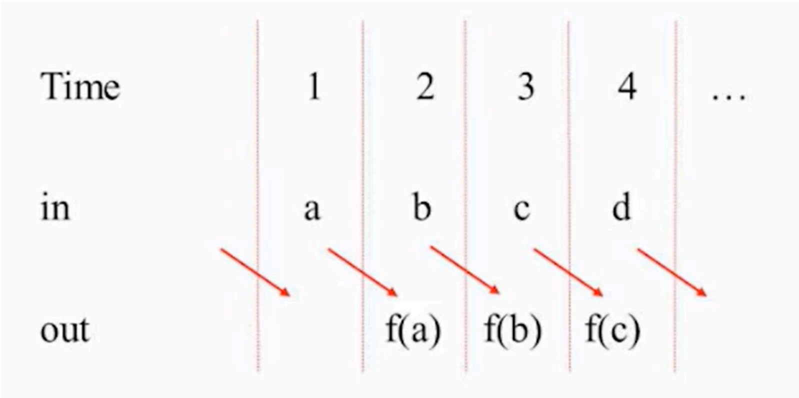
- Time intervals should be wide enough to allow for the system to stabilise

In essence,

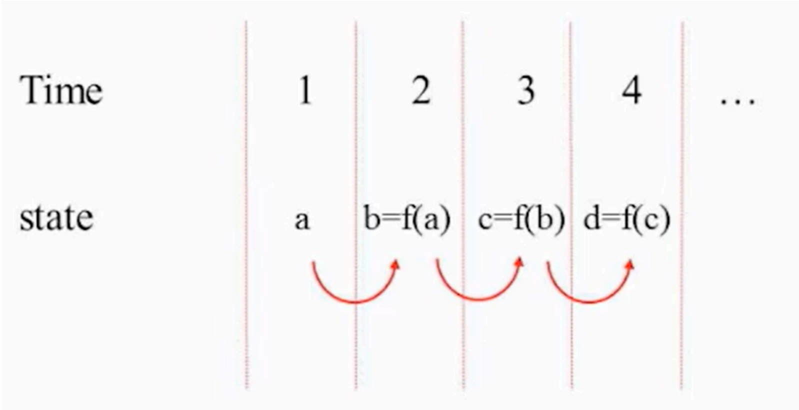
- Combinatorial: `out[t] = function( in[t] )`



- Sequential: `out[t] = function( in[t-1] )`



  - In fact, this allows us to store different values at the same place over time

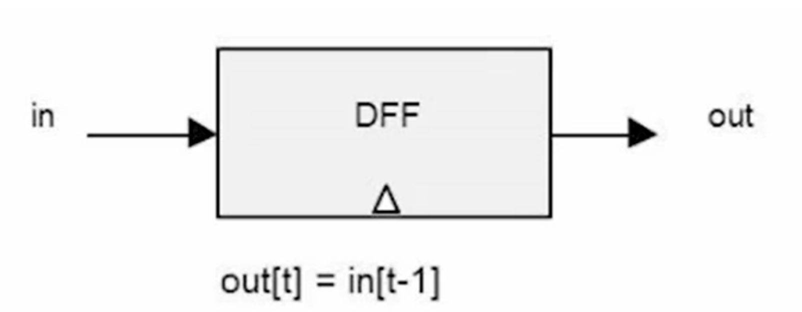


Flip-Flops

To have sequential logic, we need something that can "remember" 1-bit of information from time $t-1$ to time t.

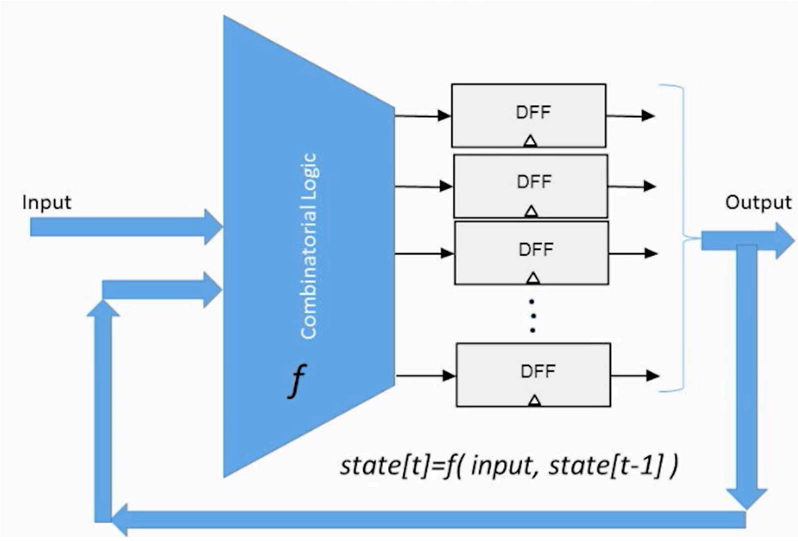
- These gates remember by "flipping" between the states 0 or 1.
- Gates that can flip between two states are called Flip-Flops.

"Clocked Data Flip Flop"

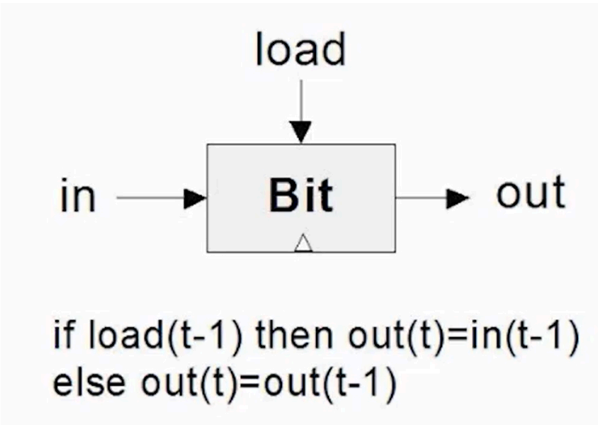


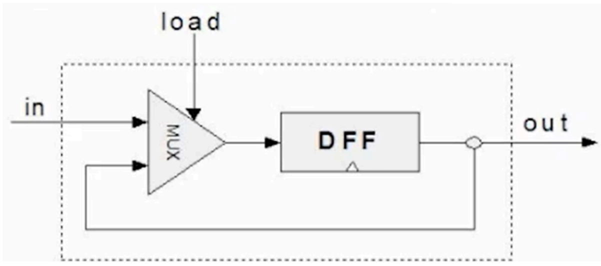
- In physical implementations, it may be built from actual Nand gates:
  - Step 1: Create a "loop" achieving an "unclocked" flip-flop
  - Step 2: Isolation across time steps using a "master-slave" setup :/
- The mini triangle △ means this is a sequential logic chip.

Sequential Logic Implementation

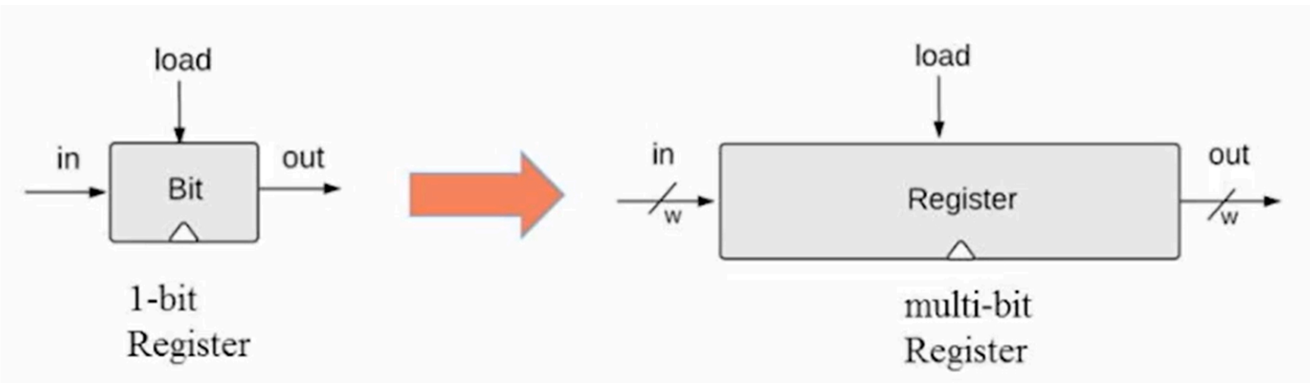


1-bit Register





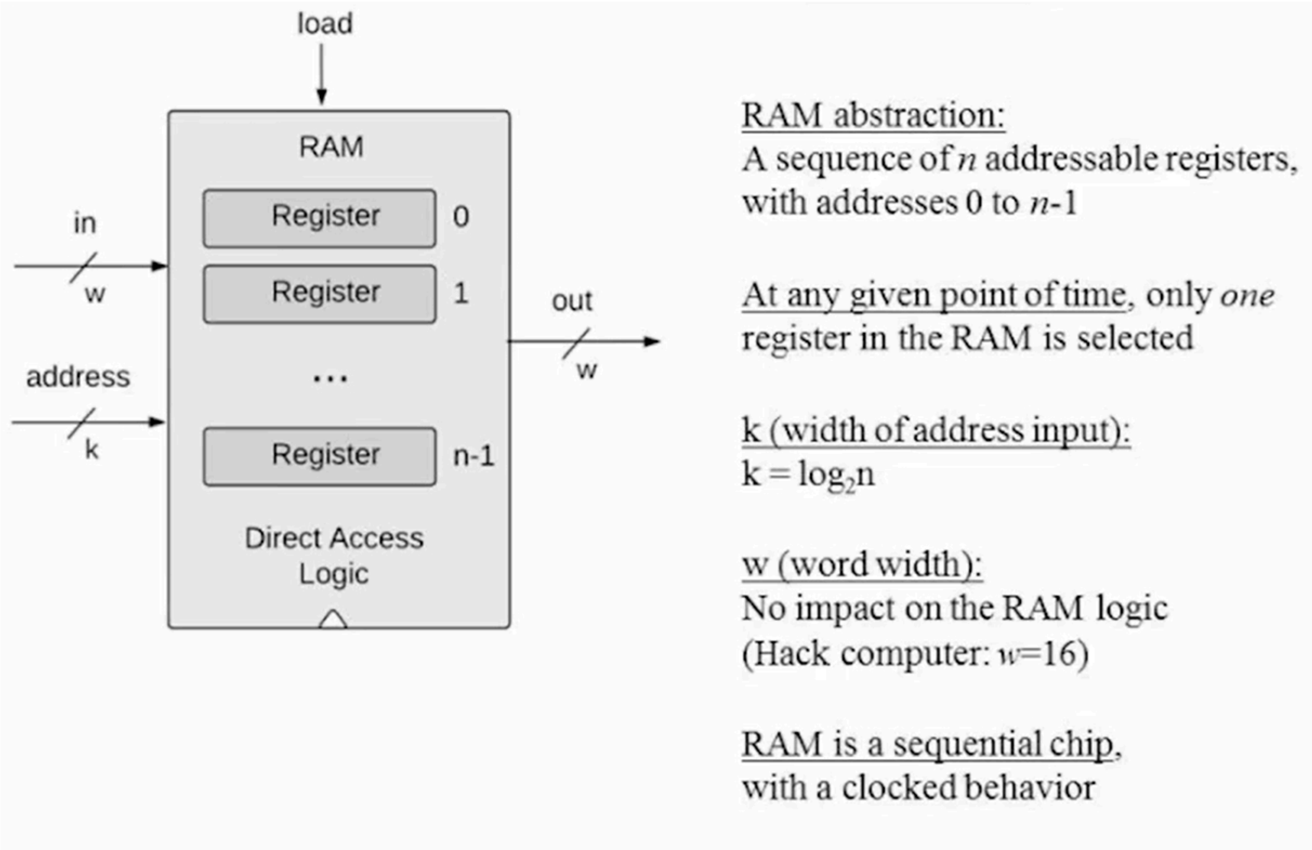
Register



- Read logic
  - Just read the output since `out` outputs register's state by default
- Write logic
  - Set `in = {desired new state}`

- Set `load = 1`

RAM unit

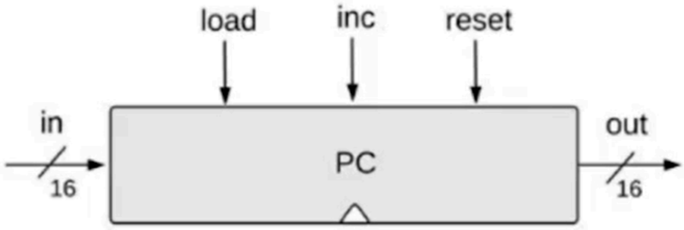


- Read Logic:
  - Set `address = {address if desired data, e.g. i}`
  - Then `out` emits the state of `Register i`
- Write Logic:
  - Set `address = {address if desired data, e.g. i}`
  - Set `in = {data}`
  - Set `load = 1`
  - Then state of `Register i` will become data

Why is it called "Random Access Memory"?

- Because irrespective of the RAM size, every register can be accessed at the same time!

Counters



PC.hdl

```
/**
 * A 16-bit counter with load, inc, and reset control bits.
 *
 * if (reset[t]==1) out[t+1] = 0                       // resetting: counter = 0
 * else if (load[t]==1) out[t+1] = in[t]               // setting counter = value
 *      else if (inc[t]==1) out[t+1] = out[t] + 1      // incrementing: counter++
 *           else out[t+1] = out[t]                    // counter does not change
```