# Sequential Logic

Programming Language Analogy:

→ One statement = Combinational Logic

eg. $Sum = f(A, B, C)$

$Sum = A + B + C$

→ Multiple statements = Sequential Logic

eg.
```
Sum = A + B;
Sum = Sum + C;
Sum = Sum + D;
```
} 3 steps

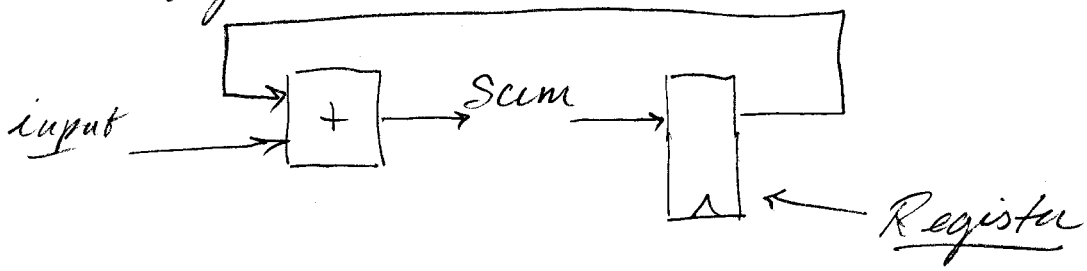
Time/Space Tradeoff:

Combinational: fast, expensive

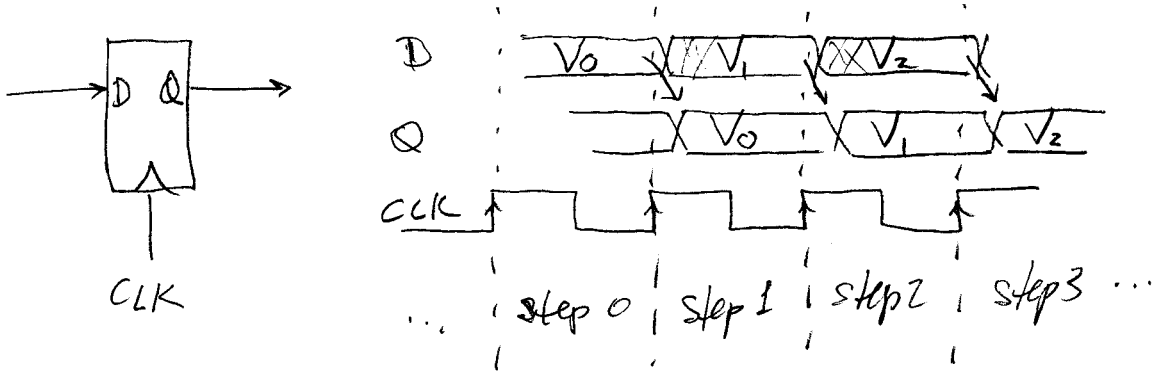
Sequential: slower, cheaper

* Neat way to save output of one step to be used on the next step.

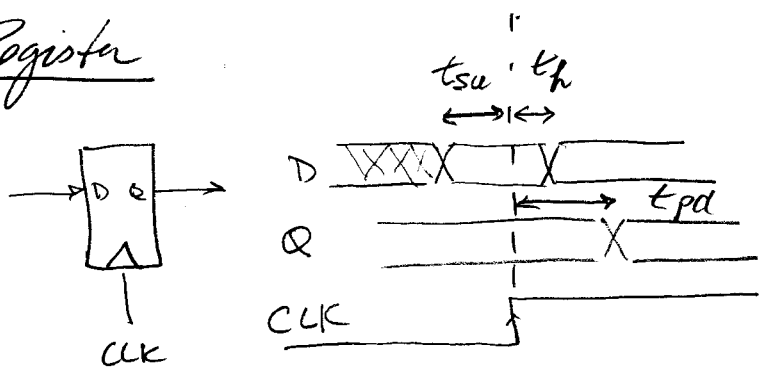
⇒ Register




Abstraction



## Register




Close-up of clock edge

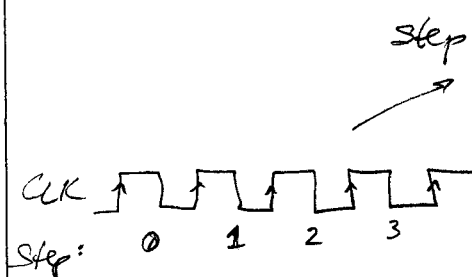
$t_{su}$ = Setup time
$t_h$ = hold time
} must obey

$t_{pd}$ = "clk-to-Q", propagation delay.

---

* => All registers sample inputs simultaneously

Example

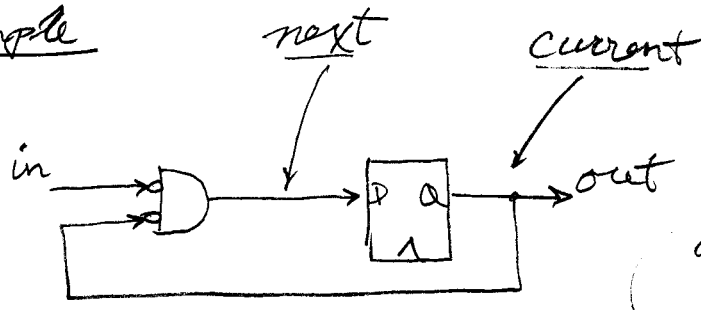



---

More complex shift registers:

* PARALLEL LOAD
* SHIFT / NO SHIFT
* RIGHT / LEFT
* PARALLEL OUTPUT

## Example

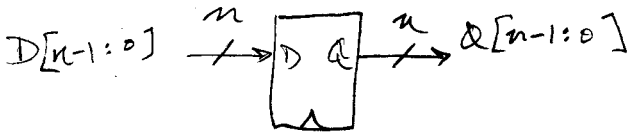



clock ↑ causes current ← next

| step | in | current | next |
|---|---|---|---|
| 0 | 1 | ? | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |

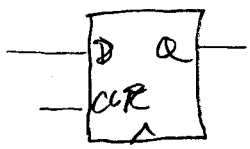
State Diagram

## n-bit register




≡

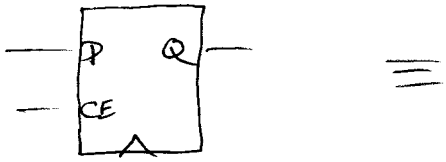
## Register with Reset




≡

## Enabled Register

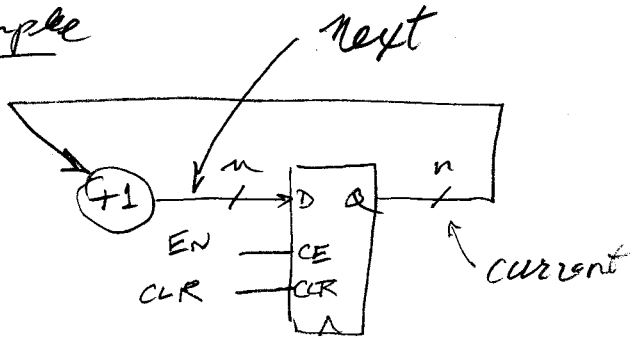
Register samples on CLK ↑ only when CE == 1.




≡

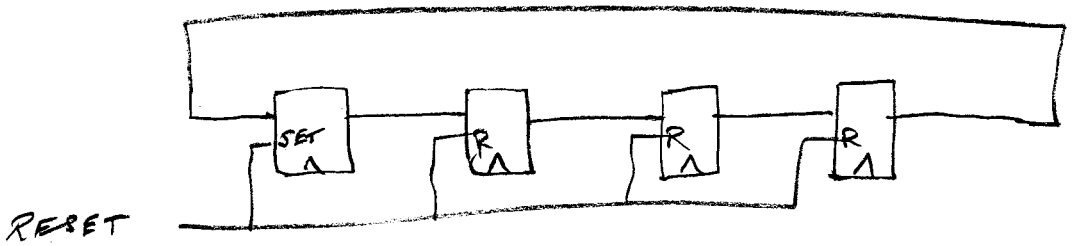
---

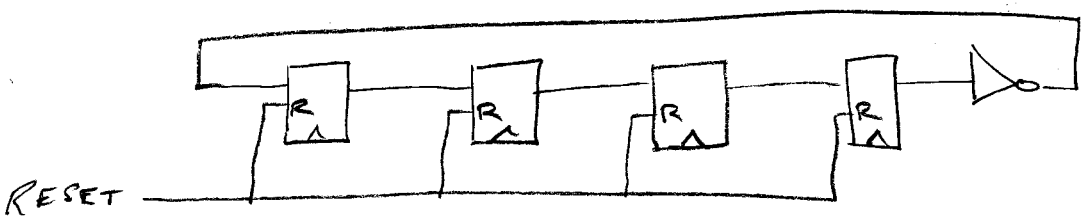
## Example

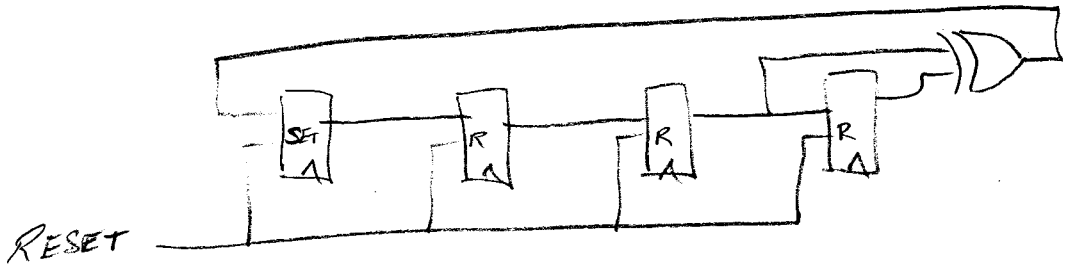



| | | CLR | EN | current | next |
|---|---|---|---|---|---|
| step | 0 | 1 | 0 | 8 | |
| | 1 | | | | |
| | 2 | | | | |
| | 3 | | | | |
| | 4 | | | | |
| | 5 | | | | |
| | 6 | | | | |

# "COUNTERS"




1-BIT BINARY COUNTER

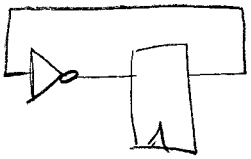
2-BIT BINARY COUNTER

3-BIT BINARY COUNTER

N-BIT BINARY COUNTER

# BINARY COUNTERS

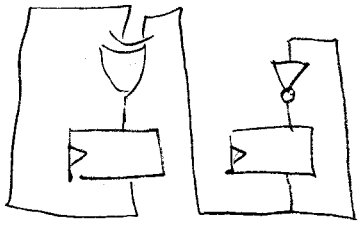
## 1 BIT



clk

| $Q^t$ | $Q^{t+1}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |

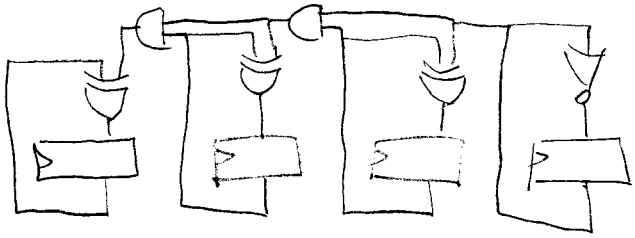
## 2 Bits



| $Q^t$ | $Q^{t+1}$ |
|---|---|
| 0 0 | 0 1 |
| 0 1 | 1 0 |
| 1 0 | 1 1 |
| 1 1 | 0 0 |
| 0 0 | 0 1 |

. . .

## 3 Bits



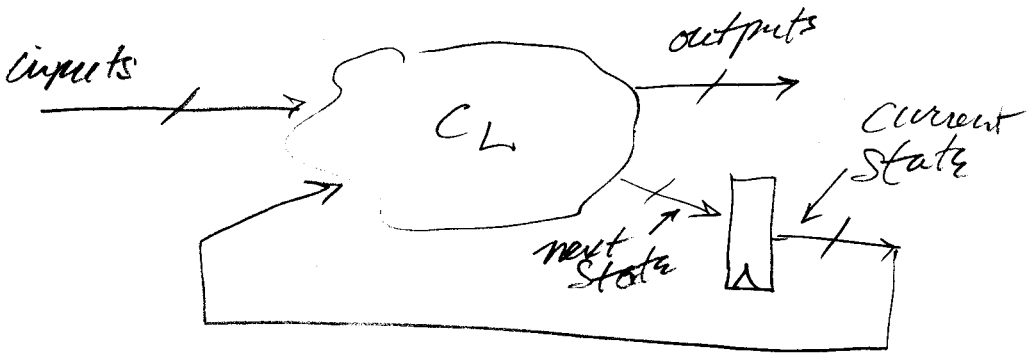
# State Diagrams

Graphical representation of a Sequential Circuit.
Registers contain the STATE.

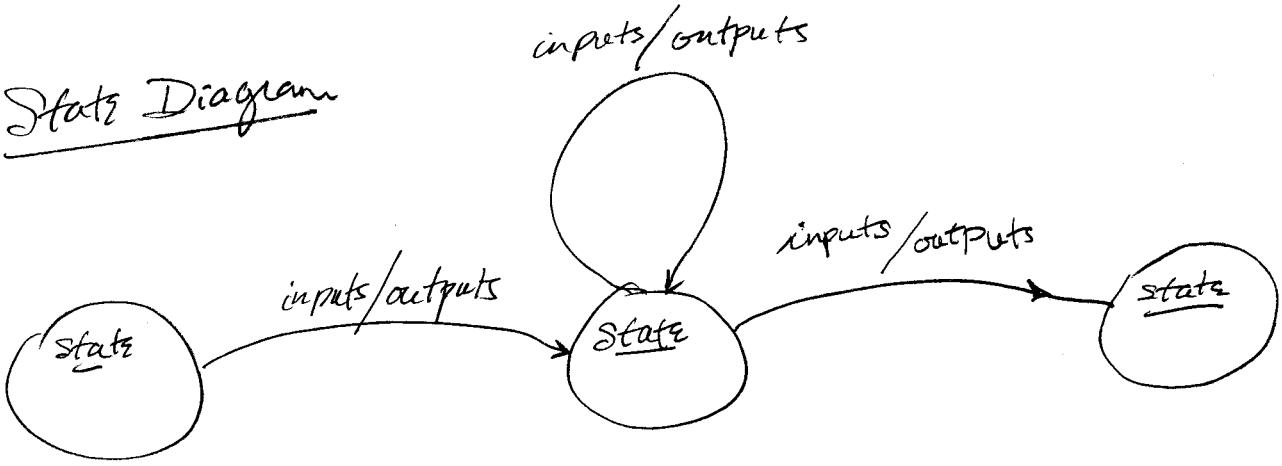
n registers ⇒ $2^n$ states

Circuit may do something different in each different state.


Combinational




## State Diagram




Arrows: Change in state caused by clock tick.

Each state has _____ out-going arrows.

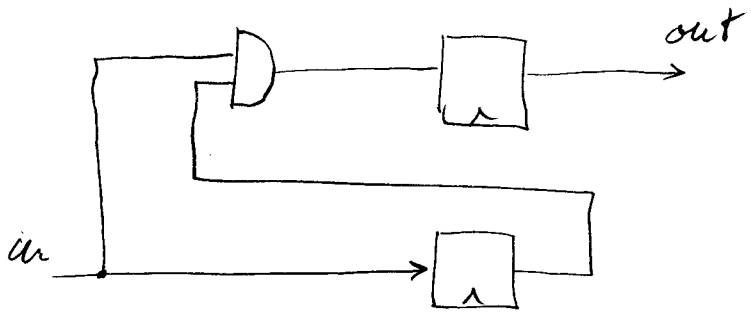
## State Transition

## State Diagrams (cont)

Given input order / output order

Reduce # of arcs by using X's for inputs

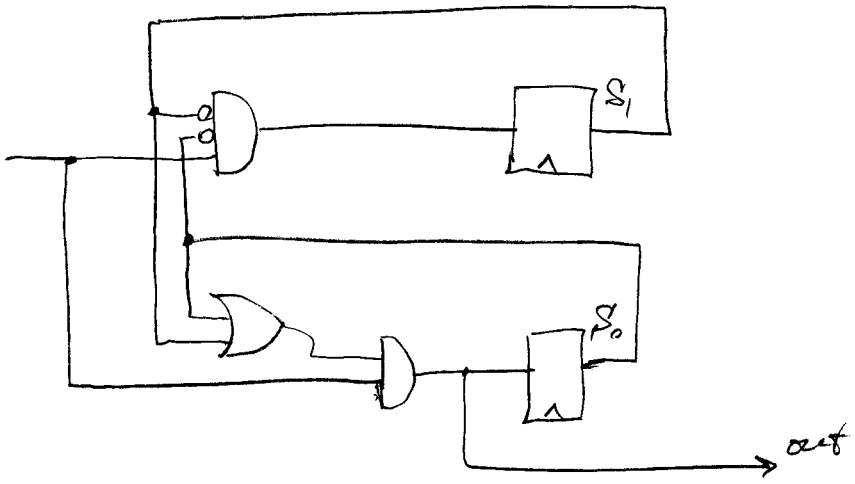
### Example




State Diagram

State Table

# Example




## State Table

| | Current | | next | | |
|---|---|---|---|---|---|
| in | $S_1$ | $S_0$ | $S_1^+$ | $S_0^+$ | out |

## State Diagram

ANALYSIS

SYNTHESIS

Design Problem

A circuit has two inputs A, B, and one output F.

A, B, F start at 0.

The circuit waits for A=1, then B=1, and then turns on F.

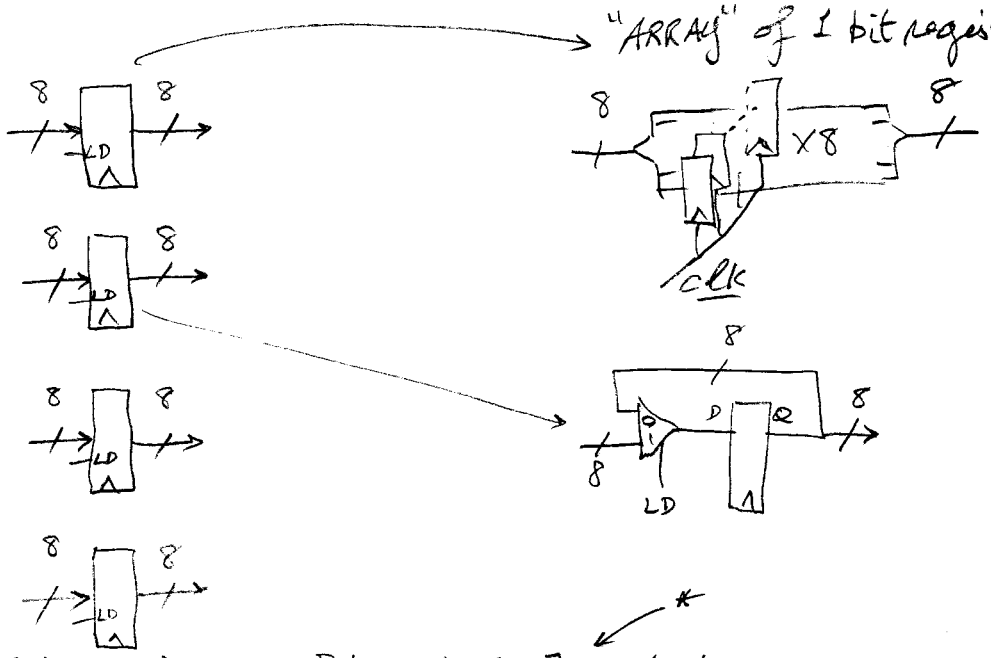
If B=1 happens before A=1, F is never turned on

# MEMORY (RAM) (1-D Array, Vector)

N locations, M bits per location (word size)
Address: $\log_2 N$ bits

$\Rightarrow$ N M-bit registers

eg 4 locations, 8-bit words.




WRITE: if(Write) MEM[WriteAddress] <= WriteData;  *

Each clock we can write a value into 1 location/register.

WriteAddress —($\log_2 N$)→ specifies which register

WriteData —(M)→ gives the data to be stored.

Write —→ control specifies whether to write

How do we implement the WRITE function?

READ: ReadData = MEM[ReadAddress];  *

We can read one location at a time. (only look at it)

ReadAddress —($\log_2 N$)→ specifies which register.

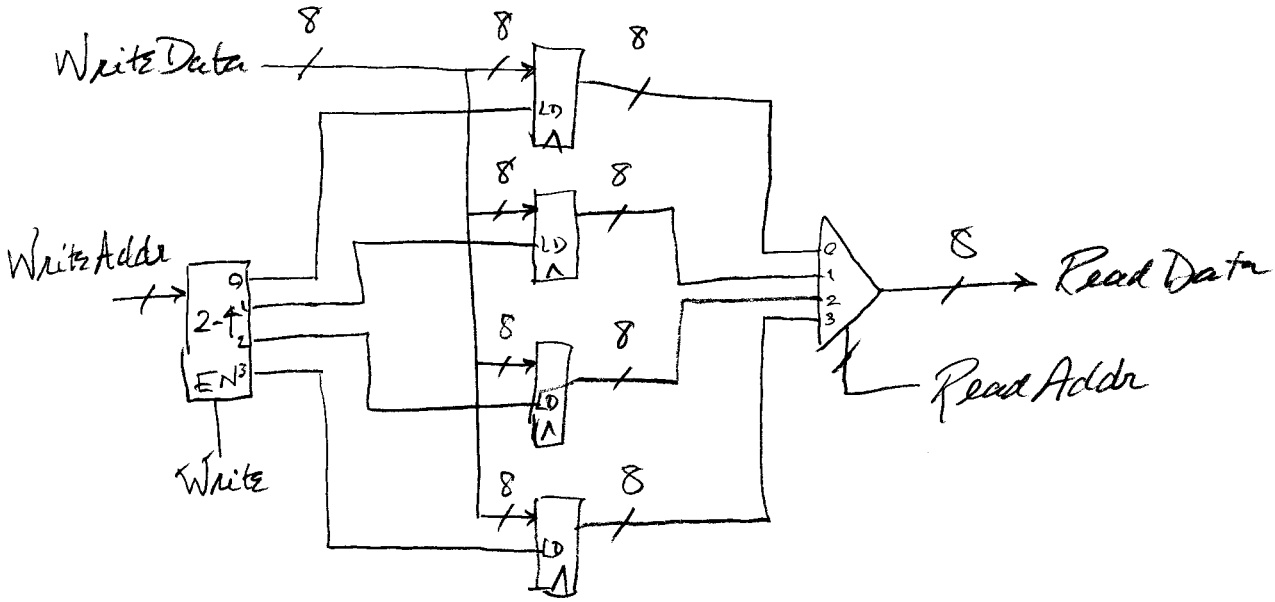
ReadData ←(M)— Data in register

NON-Destructive $\Rightarrow$ No Read control.

How do we implement the READ function?

locations → 4 × 8 ← Word size

Example: 4 × 8 Memory

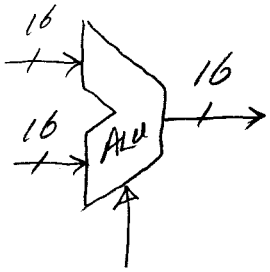



Extend to any size memory!

## PROCESSOR

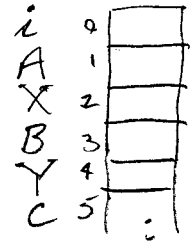
Each instruction performs an operation on data.

```
A = B+C;
Z = X-Y;
i = i+1;
```




* Variables are stored in memory.
* Each variable is allocated to a location.

eg.




Compiler does this assignment:

A = B+C ⇒ M[1] ← M[3] + M[5];

1 MACHINE instruction

For each instruction:

Read 2 values from Memory
Perform operation
Write result back to Memory