Digital Electronics

Definition
Abstraction is the hiding of data/complexity when they are not important, only expose the things needed to impletment a particular task.

Layers of Abstraction

Layer	**Examples**
Application software	Programs
Operating systems	Device drivers
Architecture	Instructions, registers
Microarchitecture	The way the processor implements the architecture
Logic elements	Adders, memories
Digital circuits	Gates and stuff
Devices	Transistors
Physics	Electrons, quantum mechanics, Maxwell's equations

This course starts at the **devices** layer.

To design something on the stack, you need to know the layer below to implement it, and the layer above to see how the thing you design is used.

Combinatory Logic

Definition
Combinatory output depends only on current input.
- It has no memory.
- The same input always gives the same output, so is entirely predictable.

Boolaen algebra is used for simplifying logic so less gates are used, so is cheaper.

Logic Gates

Definition
Logical variables can only take on two values.

In electronic circuits, they are represented by
- High voltage for 1
- Low voltage for 0

Using only 2 voltages allows circuits to have greater immunity to signal corruption caused by interference.

Definitions
- **Logical circuits** have one or more inputs.
- Basic logic circuits are known as **gates**.

Logic gates are represented in **symbol**, **truth tables** and **algebra**.
- `NOT` $\overline{a}$

  - The triangle means **buffer**.
  - The circle means **complement**.
- `AND` $a \cdot b$.
  - A lot of these gates can be extended to have more than 2 inputs.
- `OR` $a + b$
- `XOR` $a \oplus b$
- `NAND` $\overline{a \cdot b}$
- `NOR` $\overline{a + b}$

Boolean Algebra Laws

`AND` takes precedence over `OR`.

`OR`	`AND`
$a + 0 = a$	$a \cdot 0 = 0$
$a + a = a$	$a \cdot a = a$
$a + 1 = 1$	$a \cdot 1 = a$
$a + \overline{a} = 1$	$a \cdot \overline{a} = 0$

Commutation

$$a + b = b + a$$
$$a \cdot b = b \cdot a$$

Association

$$(a + b) + c = a + (b + c)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Distribution

$$a \cdot (b + c + ...) = a \cdot b + a \cdot c + ...$$
$$a + (b \cdot c \cdot ...) = (a + b) \cdot (a \cdot c) \cdot ...$$

Absorption

$$a + (a \cdot b) = a$$
$$a \cdot (a + b) = a$$

Consensus Theorem

$$a \cdot b + \overline{a} \cdot c + b \cdot c = a \cdot b + \overline{a} \cdot c$$
$$(a + b) \cdot (\overline{a} + c) \cdot (b + c) = (a + b) \cdot (\overline{a} + c)$$

Technique

1. Expand each term until it includes on instance of each variable.
2. Then simplify the terms by cancelling terms.

$x \cdot y \rightarrow x \cdot y \cdot \overline{z} + x \cdot y \cdot z$ to include z.

DeMorgan's Theorem

$$\overline{a + b + c + ...} = \overline{a} \cdot \overline{b} \cdot \overline{c} \cdot ...$$
$$\overline{a \cdot b \cdot c \cdot ...} = \overline{a} + \overline{b} + \overline{c} + ...$$

Proof

First prove for 2 varibles by truth table.

a	b	$\overline{a+b}$	$\overline{a} \cdot \overline{b}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Extend to more variables by induction:

$$\begin{aligned} \overline{(a+b)+c} &= \overline{a+b} \cdot \overline{c} \\ &= \overline{a} \cdot \overline{b} \cdot \overline{c} \end{aligned}$$

Sometimes we wish to use `NAND` or `NOR` gates since they are usually simpler and faster. We can use DeMorgan's law to replace `AND` and `OR` to `NAND` and `NOR`.

Technique: Bubble Logic
- 2 consecutive bubble operations cancel.
- Bubbles change an `AND` into an `OR` when they pass through a gate (vice versa).

Logic Minification

Normal Forms

A **minterm** contains all input variables of boolean function f so f and itself both evaluate to 1.

$x\ y\ z$	f	minterm
0 0 0	1	$\overline{x} \cdot \overline{y} \cdot \overline{z}$
0 0 1	0	
0 1 0	0	
0 1 1	1	$\overline{x} \cdot y \cdot z$
…	…	…

Definition

A boolean function f can be written in **disjunctive normal form** by expressing it as the *disjunction* (`OR`) of its minterms. (Sum of products)

$$f = \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot y \cdot \overline{z} + \ldots$$

As an inverse to minterms, **maxterm** is all the input variables where f evaluates to 0.

$x\ y\ z$	f	maxterm
0 0 0	1	
0 0 1	0	$\overline{x} \cdot \overline{y} \cdot z$
0 1 0	0	$\overline{x} \cdot y \cdot \overline{z}$
0 1 1	1	

$x\ y\ z$	f	maxterm
…	…	…

Definition
Conjunctive normal form is where the function is written as the product of sums.

$$f = (x + y + z) \cdot (x + \overline{y} + z) \cdot \ldots$$

The CNF for f can be found by writing the DNF for $\overline{f}$ (maxterms of f), then use DeMorgan's laws.

$$\overline{f} = \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot \overline{z} + \ldots$$
$$f = (x + y + \overline{z}) \cdot (x + \overline{y} + z) \cdot \ldots$$

K-Maps

Both DNF and CNF are not simplified, K-maps are useful for simplifying logical expressions of up to 5 variables.

SOP (Sum of Power) Simplification

1. Convert the truth table to K-map by plotting the minterms on the table in **grey code** order, so adjacent cells are logically next to each other.

yz

x

	00	**01**	**11**	**10**
0	1	1	1	1
1			1	

2. Group the minterms, the larger the groups the better.
   - Group sizes have to be a power of 2.
   - Groups can wrap around edges.

yz

x

	00	**01**	**11**	**10**
0	1	1	1	1
1			1	

So the simplified function is $f = \overline{x} + y \cdot z$

The simplified expression is in SOP form, suitable for implementations using `AND` and `OR` gates.

POS Simplification

1. Find a SOP Simplification for $\overline{f}$.
2. Use DeMorgan's to find a POS simplification for f.

This is suitable for implementing using `NOR` gates.

Don't Care Conditions

Some combinations will never happen, we can declare those *don't care* conditions.

- They are treated as 0 or 1 depending on which gives the simpler results.
- We write them as **`X`** in K-map.

Definitions

Keyword	Meaning
Cover	A term **covers** a minterm if the minterm is part of the term.
Prime implicant	A term that cannot be further combined.
Essential prime implicant	A term that covers a minterm that no other prime implicant covers.
Covering set	A minimu set of prime implicants which covers all minterms.

)

Q-M Method

1. List all minterms and *don't care* terms, group them by the number of **1**s.
   - Minterms: 4, 5, 6, 8, 9, 10, 13
   - *Don't care* terms: 0, 7, 15

```
0000

0100
1000

0101
0110
1001
1010

1101
0111

1111
```

2. If two terms differ by 1 digit (e.g. `0000` and `0100`), merge them and write it on the next column (e.g. `0-00` where "`-`" is a wildcard), put a tick next to them. Merge all terms that can be merged, including previously ticked terms.

   Repeat for the 2nd column, and write the merge terms to the 3rd column, until no more terms can be merged.

Column 1	**Column 2**	**Column 3**

`0000` ✓	`0-00`
	`-000`
		`01--`
		
		`-1-1`

`0100` ✓
`1000` ✓
	`010-` ✓
	`01-0` ✓
	`100-`
	`10-0`	

`0101` ✓ `0110` ✓ `1001` ✓ `1010` ✓	`-101` ✓ `01-1` ✓ `011-` ✓ `1-01`	
`1101` ✓ `0111` ✓	`11-1` ✓ `-111` ✓	
`1111` ✓		

3. The unmergable terms (no ticks beside them) are the **prime implicants**. We need to get rid of some of those terms to find the **mincover**.

   Create an **implication chart** with the **prime implicants** and the **minterms**. Put an `X` where the minterm can be expressed as the prime implicant. (e.g. $4_{10} = 0100_2$ can be expressed as `0-00`)

	4	5	6	8	9	10	13
`0-00`	X						
`-000`				X			
`100-`				X	X		
`10-0`				X		X	
`1-01`					X		X
`01--`	X	X	X				
`-1-1`		X					X

4. Look for **essential prime indicants**: 6 is only covered by `01--`, and 10 only by `10-0`, they are essential prime indicants. Cross the two row out, also cross out the numbers they cover.

	4	5	6	8	9	10	13
`0-00`	X						
`-000`				X			
`100-`				X	X		
`10-0`				X		X	
`1-01`					X		X
`01--`	X	X	X				
`-1-1`		X					X

5. Cross out as few rows as possible to cover the remaining (uncovered) numbers.

	4	5	6	8	9	10	13
`0-00`	X						
`-000`				X			
`100-`				X	X		
`10-0`				X		X	
`1-01`					X		X
`01--`	X	X	X				
`-1-1`		X					X

   The prime implicants we picked are `10-0`, `1-01` and `01--`, giving the simplification.

$$f = a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{c} \cdot d + \overline{a} \cdot b$$

Binary Adders

We are doing the **compositional approach**, where we put half adders and full adders together to build a **ripple carry adder**.

Half Adder

Adds two bits together.

By inspection or simplifying minterms:

$$\text{sum} = a \oplus b$$
$$c_{\text{out}} = a \cdot b$$

a	b	c_{out}	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

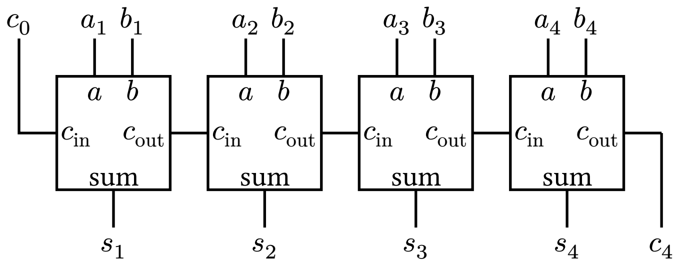
Full Adder

$$\text{sum} = a \oplus b \oplus c_{\text{in}}$$
$$c_{\text{out}} = a \cdot b + c_{\text{in}} \cdot (a + b)$$

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ripple Carry Adder

Cascade multiple of these adders together to make a ripple carry adder.



A 4 bit adder has 4 full adders.

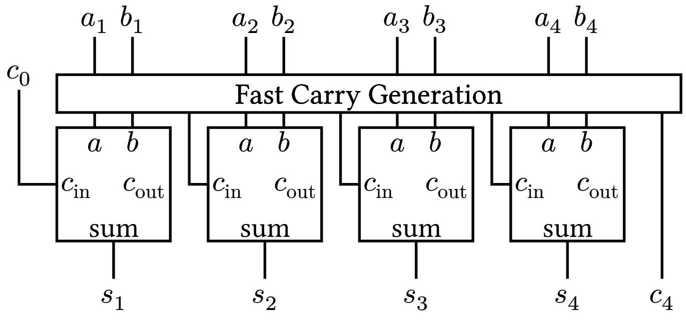
- c_0 propagates from left to right.
- Physically, they have finite propagation delay, so there is a delay between the inputs and outputs.

If we complement a then set c_0 to 1, we have $s = b - a$.

Speeding Things Up

- Design the adder as a single block of 2-level combinational logic with $2n$ inputs and n outputs.
  - Low delay.
  - Truth table is massive, and simplification is complicated. It is very complex to design.

Fast Carry Generation



c_{in}	a	b	c_{out}
?	0	0	0
?	0	1	?
?	1	0	?
?	1	1	1

- If $\overline{a_i} \cdot \overline{b_i} = 1$ then $c_{\text{out}} = 0$
- If $a_i \oplus b_i$ then $c_{\text{out}} = c_{\text{in}}$
- If $a_i \cdot b_i$ then $c_{\text{out}} = 1$

The fast carry generation has less levels than the ripple adder (think of SOP form, which only has two levels), therefore having less propagation delay.

- Lot's of gates are used in carry generation.
- Usually two 4-bit carry generators are used in an 8-bit adder.

Multilevel Logic

The ripple adder is a multilevel logic - as it cascades, it forms many levels.

- It's difficult to buy large logic gates, we can use multilevel logic to expand the input.
- But it can introduce delays and **hazards**.

Reducing Number of Gates

Simplified SOP expressions requires a lot of gates to do.

The logic below uses 9 literals, 7 gates, 2 levels.

$$\begin{aligned} z &= a \cdot d \cdot f + a \cdot e \cdot f + b \cdot d \cdot f + b \cdot e \cdot f + c \cdot d \cdot f + c \cdot e \cdot f + g \\ &= (a \cdot d + a \cdot e + b \cdot d + b \cdot e + c \cdot d) \cdot f + g \\ &= (a + b + c) \cdot (d + e) \cdot f + g \end{aligned}$$

We have recursively factored out common literals and express z in **two-level form**.

$$\begin{aligned} x &= a + b + c \\ y &= d + e \\ z &= x \cdot y \cdot f + g \end{aligned}$$

It uses 9 literals, 4 gates, 3 levels. The more levels the bigger the propagation delay.

Hazards

Definition
Hazards are brief changes in output (when there is a change in input), also called a **glitch**.

Type	Description
Static hazard	Output undergoes momentary transition when one input changes when it is supposed to remain unchanged.
Static 1 hazard	Output should stay at 1, but as signal changes it briefly drops to zero.
Static 0 hazard	The opposite of a static 1 hazard.
Dynamic hazard	Output changes more than once as opposed to just once.

This is due to the difference in **propagation delay** between layers.

Removing Hazards

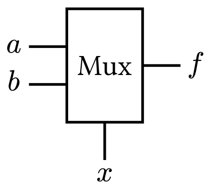
1. Plot the function on a K-map.
2. If the square it is moved from and to are not in the same group, there is a hazard.
3. Add an extra term containing the two squares to remove the static 1 hazard.

To remove a static 0 hazard, draw a K-map of the complement of the output.

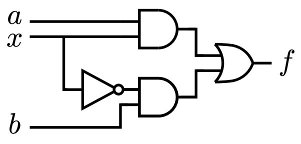
Multiplexers

Definition
Multiplexers (mux/selector) chooses 1 of many inputs to output according to the control input.

2-to-1 Selector



a	b	x	f
X	b	0	b
a	X	0	a



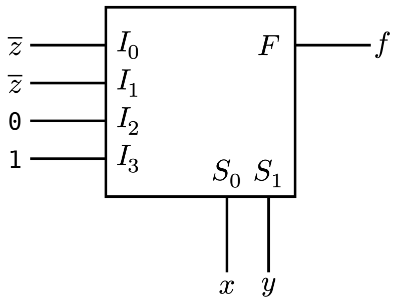
You can also express it as the sum of minterms with using a full truth table.

- n-to-1 mux is possible - an 8:1 mux requires 3 control lines.
- The mux is also called a hardware lookup table.

Sometimes it is possible to use less control lines if we use the logic variables as control input.

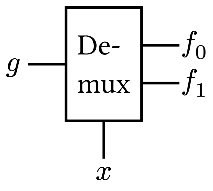
a	b	x	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

xy	f
00	$\overline{z}$
01	$\overline{z}$
10	0
11	1

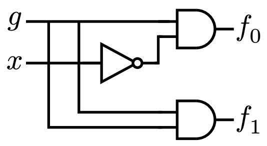


Demultiplexer

A single output is directed to exactly one of its inputs.



g	x	f_0	f_1
g	0	g	0
g	1	0	g



Larger demultiplexers are possible, e.g. a 3:8 demux.

A **decoder** is a demux where g is permanently set to 1, so only one output is 1 at any time. An **enabler** enables 1 out of n logical subsystems.

If the number of pins is limited, using a decoder/multiplexer is essential.

- Without decoder: you need 8 pins to control 8 subsystems.
- With decoder: you need 3 pins to control 8 subsystems.

We can also create any combinational logic block using the decoder. By OR-ing the outputs that corresponds to the minterms of the logic.

ROM

ROM is a storage device that can be

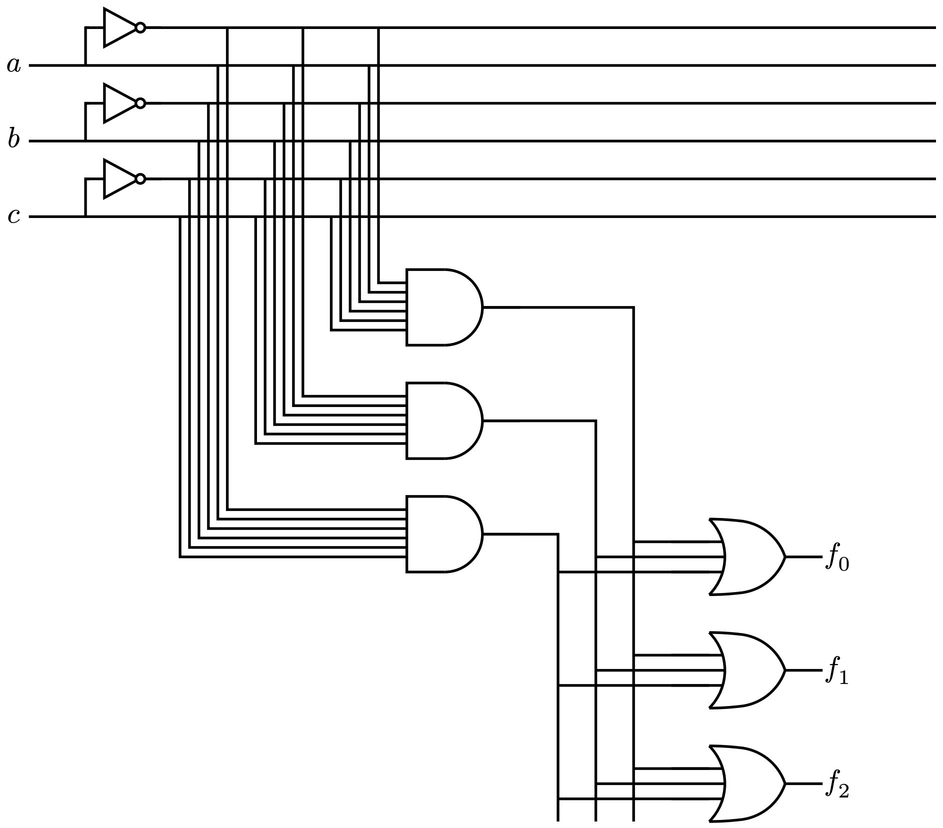
- Written into once

- Read at will
- Non-volatile
- Essentially a lookup table: n output lines specify the address of location holding m-bit data words.

The ROM has 2^n possible locations.

We can create any combinational logic by holding all the minterms in ROM and output the stored value.
- No simplification needed.
- Reasonably efficient if lots of minterms need to be generated.
- Can be inefficiently large if many spaces are zero (there are very few minterms).

Programmable Logic Array
In a PLA, only the required minterms are generated using the **AND-plane** and **OR-plane**.



The PLA is programmed by selectively removing connections from the AND-plane and OR-plane.

Programmable Array Logic
To simplify design, the OR-plane is not programmable. There are instead multiple AND-planes each connected to an OR gate.

Memory Applications
Other memory devices includes:
- Non-volatile memory by ROMs and flash.
- Volatile memory offered by static RAM or dynamic RAM (much denser than SRAM, but has to be refreshed regularly).

Memory is connected to the CPU using **buses**.

Definition
Buses are a bunch of wires in parallel.

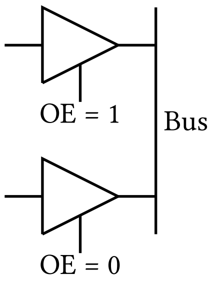
- The **address bus** specify the memory location being accessed.
- The **data bus** conveys data to and from that location.

Using Multiple Memory Devices

More than 1 memory device can be connected to the same bus wires. **Tristate buffers** controls which devices to enable by disconnecting the device from the bus when not selected.

The tristate buffers are controlled by **output enabled** (OE) control signals, the other control signals are:

- **Write enable** (WE) determines whether data is written or read.
- **Chip select** (CS) determines if the chip is activated, otherwise the chip is powered down to conserve power.



Sequential Logic

... is the end to combinational logic.

- **Combinational logic** depends only on the condition of the latest inputs.
- **Sequential logic** also depend on earlier inputs.

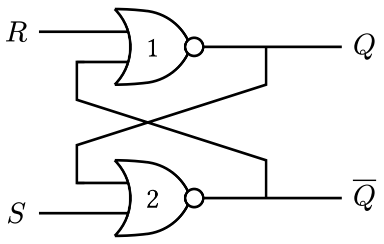
Definitions
- **Memory** stores data from earlier.
- A snapstop of memory is called **state**.
- 1 bit memory is called a **bistable**.

Flip-flops and **latches** are are implementations of bistable.

RS-Latch

The RS-latch is a memory element with 2 inputs. Where Q is the current state, and Q' the next state.

S	R	Q'	$\overline{Q}$	**Comment**
0	0	Q	$\overline{Q}$	hold
0	1	0	1	reset
1	0	1	0	set
1	1	X	X	illegal



- Consider $R = 1$ and $S = 0$:
  - Gate 1 is 0, so Q is 0.
  - Gate 2 gives complement of gate 1, so 1.
- Consider $R = 0$ and $S = 0$:
  - Gate 1 gives the complement of gate 2.
  - Gate 2 gives the complement of gate 1, which is the hold condition.

State Transition Diagrams

1. Create a truth table of the RS-latch.

Q	S	R	Q'	Comment
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	0	illegal

Q	S	R	Q'	Comment
1	0	0	1	hold
1	0	1	0	reset
1	1	0	1	set
1	1	1	0	illegal

2. Consider all cases where $Q = 1$ and $Q' = 1$.

Q	S	R	Q'
1	0	0	1
1	1	0	1

So when $Q = 1$, the next state $Q' = 1$ if

$$\overline{S} \cdot \overline{R} + S \cdot \overline{R} = \overline{R}$$

3. Repeat this for all other state transitions, we get.

$$\begin{tikzcd}
Q=0 \arrow[loop left, "\overline{S}+R"] \arrow[r, bend left, "S\cdot\overline{R}"] & Q=1 \arrow[loop right, "\overline{R}"] \arrow[l, bend left, "R"]
\end{tikzcd}$$

Transparent D-Latch

The output of a transparent D-latch will only change when there is a global **enable signal** called the system clock. This makes it easier to design large scale systems, as opposed to letting the circuits change state when ever the input changes.

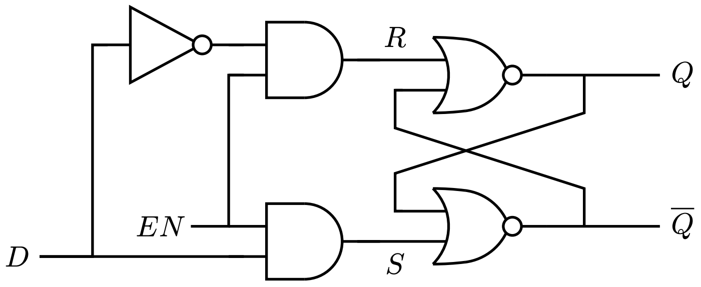
Definition

The **system clock** produces square wave signals at a frequency.

- Imposes order on state changes.
- Allows a lot of states to update simultaneously.

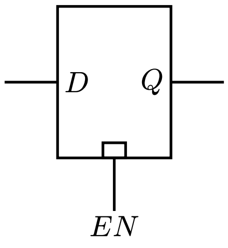
Modify the RS-Latch so the signal only change there is an enabling signal.

- There is only 1 input, R and S are complement of each other, so we can never set both of them to 1, avoiding the unwanted combination.



The square box means it is **level activated**. When it is enabled, it is in **transparent mode**.

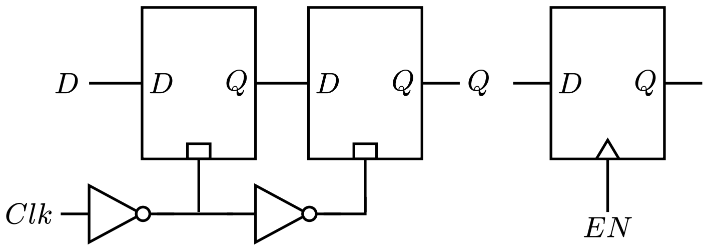
- $Q = D$ if $EN = 1$
- Q remains in previous state if $EN = 0$



Master Slave Flip-Flops

The output changes only on a rising edg.

- It is much simpler to design sequential circuits if outputs only changes on rising or falling edges.
- In lab we use a truly F-F device (not affected by propagation delay).



Other Types of Flip-Flops

- The **JK F-F**: a clocked RS F-F that has 2 outputs Q and $\overline{Q}$. The illegal state now represents **toggle**.
- The **T F-F**: similar to the D F-F but there are 2 outputs.

Asynchronous Inputs

It is common for F-Fs to have additional asynchronous inputs which are independent of clock signals.

- Reset/clear: set Q to 0
- Preset/set: set Q to 1

Timing

The transparent D-latch requires

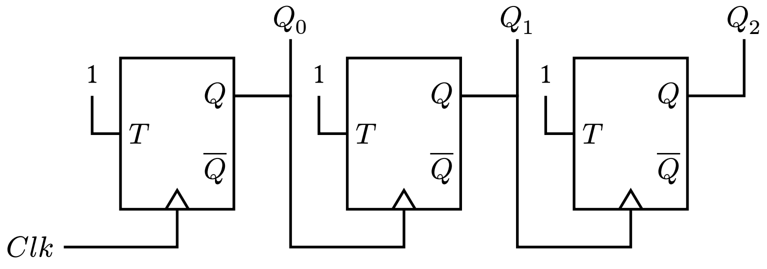
- D to change before a minimum **setup time** t_{su} before the clock signal.
- The data is held for **hold time** t_{h} after the clock edge.
- Then the output will change in **propagation delay** t_{p} after the clock edge.

Flip-Flops Applications

Definition

A **counter** is a clock sequential circuit that goes through a predetermined sequence of states.

A **ripple counter** can be made by cascading edge triggered T-type F-Fs operating in toggle mode.



A string of n F-Fs can have 2^n states.

Counters are used for

- Counting.
- Producing a delay of a particular duration.
- Generating sequences.
- Dividing frequencies by 2.

A ripple counter is not a synchronous device as the clock to the next stage comes from the previous stage, in a true synchronous device, all the clock input to the flip flops come from the clock.

- Output does not change synchronously - hard to know when the output is actually valid.

- Propagation delay bulids up, limiting the maximum clock speed before **miscounting** occurs. Where Q_0 changes to the next state before Q_2 manages to change to the current state.

BCD Counters

To count a number that is not a power of 2 (e.g. 10), we need
- An F-F with a clear/reset signal.
- An AND gate to detect the count of 10 and use its output to reset the F-F.

Synchronous Counters

We can identify a synchronous design if all F-F clock inputs are connected to the clock signal, so they all change at the same time.

The **excitation table** shows the input required for a particular transition. Note that D is the input we want to create, and Q' is caused by D.

$Q_2Q_1Q_0$	$Q_{2'}Q_{1'}Q_{0'}$	$D_2D_1D_0$
000	001	001
001	010	010
010	011	011
011	100	100
100	101	101
101	110	110
110	111	111
111	000	000

By inspection, we can see that $D_0 = \overline{Q_0}$ and $D_1 = Q_1 \oplus Q_0$.

To find out D_2, we need to draw a K-map.

Q_1Q_0

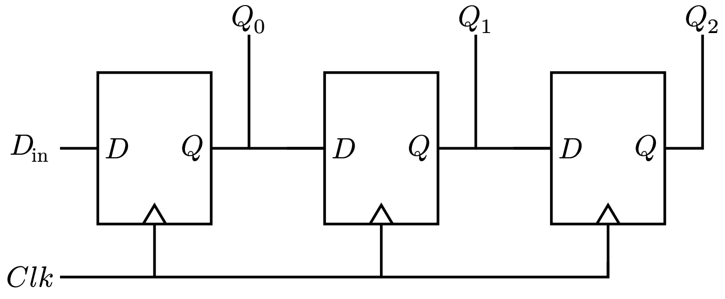
Q_2

	00	01	11	10
0			1	
1	1	1		1

$$\begin{aligned} D_2 &= Q_2 \cdot \overline{Q_1} + Q_2 \cdot \overline{Q_0} + \overline{Q_2} \cdot Q_1 \cdot Q_0 \\ &= Q_2 \cdot \left(\overline{Q_1} + \overline{Q_0}\right) + \overline{Q_2} \cdot Q_1 \cdot Q_0 \end{aligned}$$

The design process for counters that count in any sequence is the same.

Shift Registers



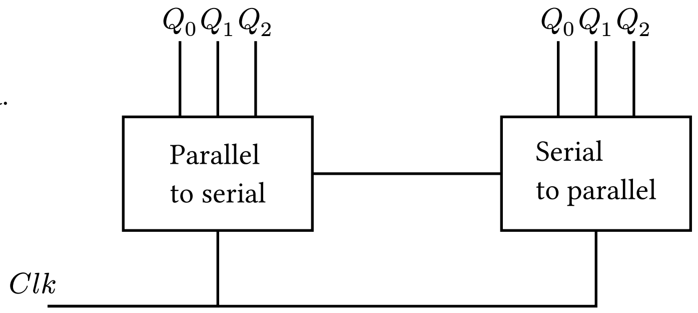
A shift register is a **synchronous machine** because all the F-Fs are connected to the same clock.
- Serial input, parallel output.
- The first clock Q_0 outputs the value of D at the next clock edge.

- Q_1 output is delayed by 1 clock cycle.
- Q_2 output is delayed by 2 clock cycles.

Application

Use it as a **serial data link**.

1. Parallel data in.
2. pass through a wire as serial data.
3. Parallel data out.



System Timing

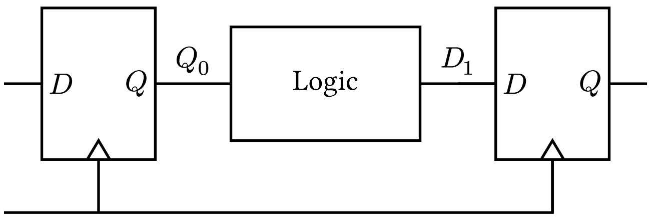
Definitions

- The **clock period** T_c is the time between rising clock edges.
- The **clock frequency** $f_c = 1/T_c$.

Setup Time Constraint

The correct operation of a D-type F-F requires

- Minimum **setup time** T_{su} before the edge.
- Minimum **hold time** T_h after the edge
- Output changes in **propagation delay** T_{pc} after the clock.



To satisfy the setup time for the 2nd F-F D_1 must settle before t_{su} before the next clock. Giving the **minimum setup period**.

$$T_c \geq t_{pc} + t_{pd} + t_{su}$$

Hold Time Constraint

For the 2nd F-F to output correctly, D_1 must hold (not update) for a minimum of t_{hold} after clock.

$$\min(t_{pc} + t_{pd}) \geq t_{\text{hold}}$$

- We would expect 2 F-Fs to cascade with no combinational logic in between, so

$$\min(t_{pc}) \geq t_{\text{hold}}$$

- Often F-F are designed with $t_{\text{hold}} = 0$.

Hold time violations cannot be fixed by adjusting clock period and is hard to fix.

Note
- Setup time constrain is concerned with the **maximum propagation delay**. (time after clock edge required to guarantee stable output)
- Hold time constraint concerned with the **minimum propagation delay**. (time after clock edge which output is guaranteed to not change)

Clock Skew

- Previously we assumed all clock signals reach all the F-Fs the same time.
- Skew is caused by the differences in the length of physical lengths from the clock to the flip-flops.

This increases the **maximum propagation delay** possible.

$$T_c \geq t_{pc} + t_{pd} + t_{su} + t_{\text{skew}}$$

- Skew decreases the maximum propagation delay allowed $t_{pc} \leq T_c - (t_{pc} + t_{su} + t_{\text{skew}}$
- But makes the hold time constraint looser $\min\left(t_{\text{skew}} + t_{pc} + t_{pd}\right) \geq t_{\text{hold}}$

Meta Stability

It is not always possible to control when an input to an F-F changes (e.g. user inputs). If the **dynamic requirements** are violated:
- This causes output Q to be **undefined** (between 0V and V_{DD}).
- Q will remain in a **metastable state** (undefined) until it resolves to a stable valid state.

The resolution time is unbounded, the distribution of resolution time is given by

$$P(t_{\text{res}} > t) = \frac{T_0}{T_c} \exp\left(-\frac{t}{\tau}\right)$$

Where T_0 and τ are the characteristics of the F-F.

So the longer we wait, the less likely the F-F is still in the metastable state.

Note
P_{fail} can present **mean failures per second** and can go over 100.

Synchroniser

A synchroniser is made by cascading an extra F-F after the first F-F.
1. The probability of each F-F resolves to a valid level t_{su} before the next clock edge is
$$P_{\text{fail}} = P(t_{\text{res}} > T_c - t_{su}) = \frac{T_0}{T_c} \exp\left(-\frac{T_c - t_{su}}{\tau}\right)$$
2. For n cascaded F-Fs to resolve to a metastable state, the probability is $(P_{\text{fail}})^n$.

Mean Time Between Failures

If input D changes N times per second, the probability of failure is NP_{fail}/s.

System reliability is measured in mean time between failures.

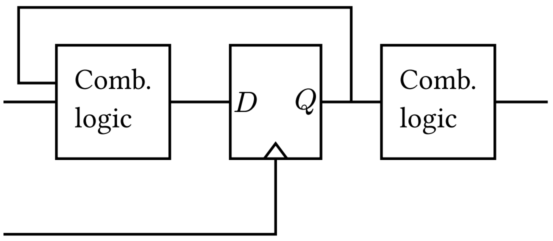
$$MTBF = \frac{1}{NP_{\text{fail}}/s} = \frac{T_c \exp\left(-\frac{T_c - t_{su}}{\tau}\right)}{NT_0}$$

Finite State Machines

Definitions

Keyword	Meaning
Finite state machine	A deterministic circuit that produces outputs which depends on its internal state and external outputs.
State	A set of internal memorised values.
Input	External stimuli to the FSM.
Output	Results of the FSM.

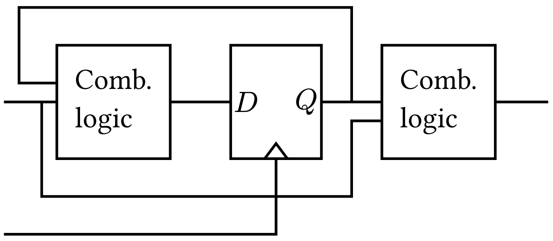
Moore machine's output only depends on the state. The 2nd combinational logic is **optional**.



The Moore machine is clocked and relatively predictable. Although the combinational logic may have a glitch.

Only input shown on transition arcs, next state is given by current state and current input.

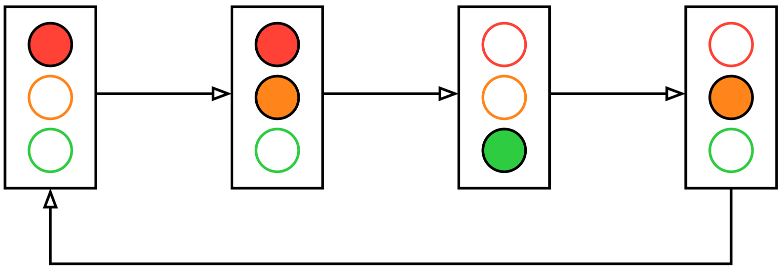
Mealy machine also additionally takes the input to generate an output directly.



We know inputs are generally not synchronised, so the combinational logic can change at any time.

Inputs and outputs shown on transition arcs, because the output is determined by current input and current state.

Example: Traffic Lights



RAG	$R'A'G'$
`100`	`110`
`110`	`001`
`001`	`010`
`010`	`100`

4 states, so minimum 2 F-Fs required, but if we have 3 F-Fs we don't need any additional combinational logic, we only use 4 out of 8 possible states.

We can use K-maps to find the expressions for R', A' and G'.

Initial State

The FSM can be powered up in any of the unused state. This may cause the FSM to be looping in an arbitrary sequence of unintended cases. We can solve that by

1. Checking if the FSM eventually enter a known state from the unused states.
2. If not, we can insert an unused state to the state transition transition table, so the next state is a used state.

State Assignment

State assignment is nontrivial, and depends on what we are trying to optimise:

- Complexity (e.g. no. of chip used, wiring complexity)
- Speed

If we have n states, we need at least $\log_2 n$ F-Fs (a.k.a. bits) to encode the states.

Sequential State Assignment

1. Assign states in increasing natural binary count.
2. Write down the state transition table using $\log_2 n$ gates.
3. Determine the next state logic and the output logit.

cba	$c'b'a'$
000	001
001	010
010	011
011	100
100	000

$$'a = \overline{a}$$
$$'b = a \oplus b$$
$$'c = a \cdot b$$

Sliding State Assignment

cba	$c'b'a'$
000	001
001	011
011	110
110	100
100	000

$$'a = \overline{b} \cdot \overline{c}$$
$$'b = a$$
$$'c = b$$

Note the simpler logic, but more unusable states.

Shift Register Assignment

$edcba$	$e'd'c'b'a'$
00011	00110
00110	01100
01100	11000
11000	10001

$$'a = e$$
$$'b = a$$
$$'c = b$$
$$'d = c$$
$$'e = d$$

Extremely simple logic, but a lot of unusable states.