Combinational logic

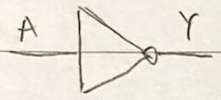
Logic gates

Logic variables and logic gates

- Logic/binary/boolean variables can only take 2 values — TRUE (1) or FALSE (0).
- In electronic circuits, the 2 states of logic variable are represented by 2 voltage levels.
- Electronic circuits that have logic signals in their i/ps and o/ps are logical/digital circuits
- Basic logic circuits w/ one or more inputs, and one output are known as gates, which can implement on an individual logic function.

NOT gate

- Graphical symbol



i/p-o/p map

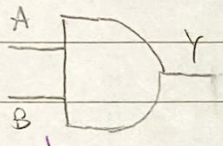
A	0	1
	1	0

Boolean representation

$Y = \bar{A}$

AND gate

- Graphical symbol



more inputs are possible

i/p-o/p map

B \ A	0	1
0	0	0
1	0	1

Boolean representation

$Y = A \cdot B$

OR gate

- Graphical symbol

more inputs are possible

i/p-o/p map

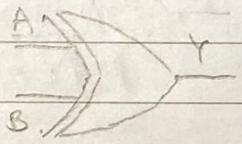
B \ A	0	1
0	0	1
1	1	1

Boolean representation

$Y = A + B$

XOR gate

- Graphical symbol



only 2 inputs

i/p-o/p map

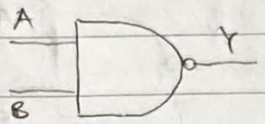
B \ A	0	1
0	0	1
1	1	0

Boolean representation

$Y = A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$

NAND gate

- Graphical representation



more inputs are possible

i/p-o/p map

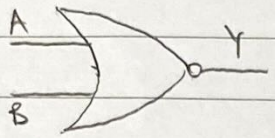
B \ A	0	1
0	1	1
1	1	0

Boolean representation

$Y = \overline{A \cdot B}$

NOR gate

- Graphical representation



more inputs are possible

i/p-o/p map

B \ A	0	1
0	1	0
1	0	0

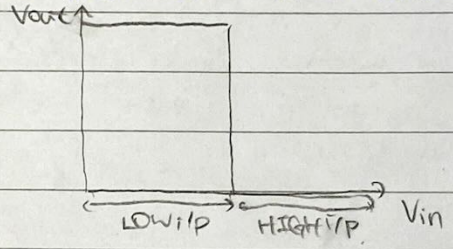
Boolean representation

$Y = \overline{A + B}$

Building gates from transistors

Ideal inverter.

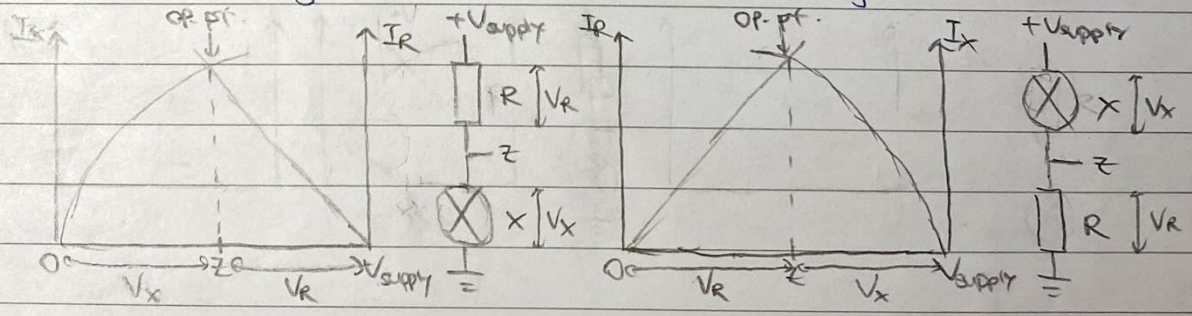
- A high voltage represents logic 1, a zero voltage represents logic 0
- The characteristics of an ideal inverter is as follows:



- This characteristic is non-linear → we can use a graphical technique for analyzing non-linear circuits.

Solving non-linear circuits.

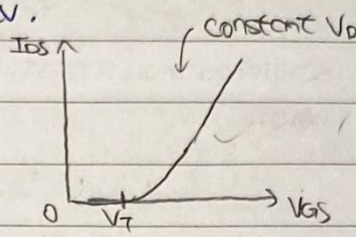
- We can use the following graphical method to solve for the operating pt.



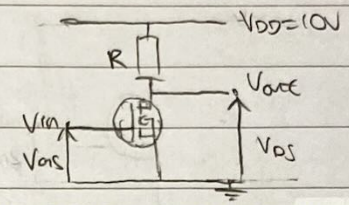
- The characteristic of the lower component is plotted normally, whereas the characteristic of the upper component is drawn backwards along the V axis, starting at V_{supply}
- The operating pt. is the intersection of the characteristics.

NMOS inverter

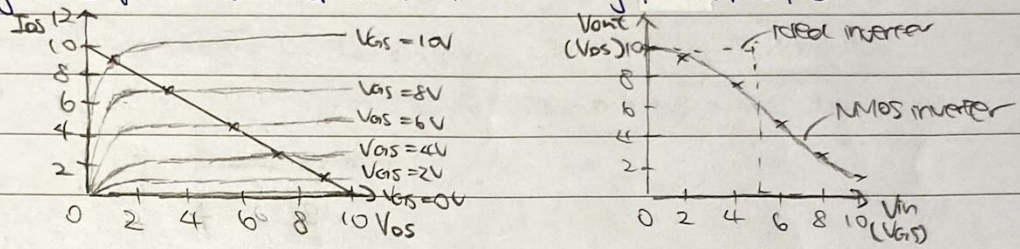
- An n-type enhancement mode MOSFET conducts when V_{GS} exceeds a threshold voltage V_T, typically about 2V.



- The circuit of a simple NMOS inverter is as shown:



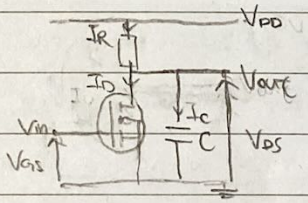
- Applying the graphical method to find the operating pt → I/p-O/p characteristic



- The input-output characteristic of the NMOS inverter is far from the ideal inverter
- However, if we say $V>9V$ is logic 1 ; $V<2V$ is logic 0, then the gate will work: $V_{in}>9V \rightarrow V_{out}<2V$; $V_{in}<2V \rightarrow V_{out}>9V$
- If we redefine the voltage for logic 1/0, the circuit characteristic does not deviate too much from the ideal

Speed of NMOS logic

- One of the main speed limitations in real NMOS logic is due to stray capacitance. We can modify the circuit as shown below.



- Assume that initially Vin is HIGH, so $V_{out}=1V$. At time $t=0$, Vin falls to 0V, so $I_D=0$

KCL: $I_R = I_C + \cancel{I_D}^{\,0} \rightarrow \frac{V_{DD}-V_{out}}{R} = C\frac{dV_{out}}{dt}$

Rearranging: $\frac{dV_{out}}{dt} = -\frac{1}{CR}(V_{out}-V_{DD}) \rightarrow \int_1^{V_{out}} \frac{1}{V_{out}-V_{DD}}\,dV_{out} = -\int_0^t \frac{1}{RC}\,dt$

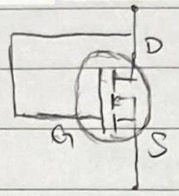
$$\therefore V_{out} = V_{DD} + (1-V_{DD})e^{-t/RC}$$

- Typically, $C \sim 10pF$ (stray) and $R \sim 1k\Omega$ so $RC = 10ns$.
- To switch (charge), we need to reach 95% V_{DD} → we need $\sim 3RC = 30ns$.
- If discharging of charge through the capacitor takes ~10ns, then a full cycle (charge+discharge) would take at least 40ns ($f = \frac{1}{40ns} = 25MHz$)

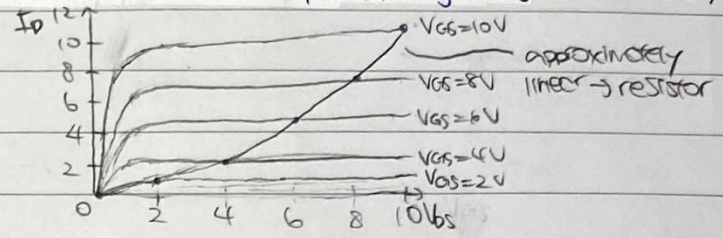
Using a FET to replace the resistor

- A big advantage of transistors is that they are small. Implementing a resistor on a chip takes up a lot of space on the silicon, (cost of chip ∝ size of chip) ← 100 × smaller than resistors

- We can make a MOSFET behave (almost) like a resistor by connecting the gate to the drain.

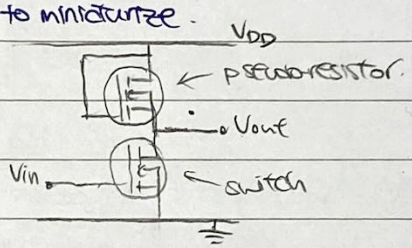


$V_{DS} = V_{GS}$.



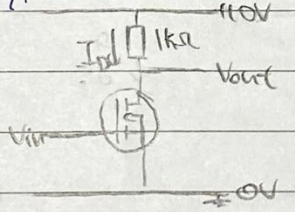
- The characteristic is not quite a straight line so its behaviour is not exactly the same as a resistor. However, we only deal w/ extrema in digital circuits, so this approximation is good enough (Resistance ≈ 900Ω).

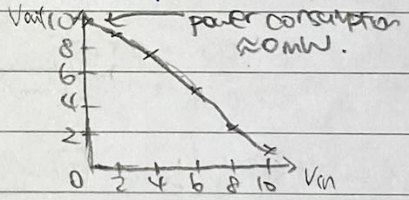
- Using a FET to approximate a resistor, we can produce a revised design for an inverter circuit that is easier to miniaturize.



Power consumption of NMOS inverter

- For simplicity, consider the NMOS inverter w/ a resistor.





- When $V_{out} = 10V$, there is no pd across the resistor, so there is no current (I_D) and hence no power is dissipated.

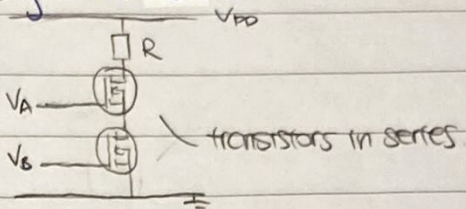
- When $V_{out} \neq 10V$, the pd across the resistor is no longer zero, so current flows down the resistor and power is dissipated as heat.

- For example, when $V_{in} = 10V$, $V_{out} = 1V$, $I_D = \frac{10-1}{1000} = 9mA$ so using $P = I_D^2 R$ and $P = I_D V$, we find that 81mW and 9mW will be dissipated in the resistor and the FET respectively.

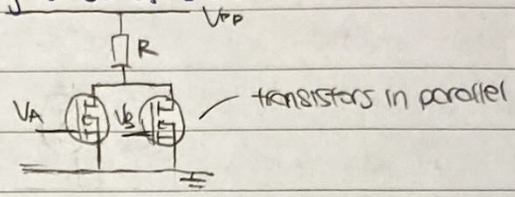
* Note that $P_{FET} \ll P_R$.

NAND gate and NOR gate using NMOS logic

- A NAND gate can be built using NMOS as shown below:

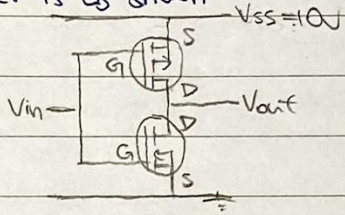


- A NOR gate can be built using NMOS as shown below:

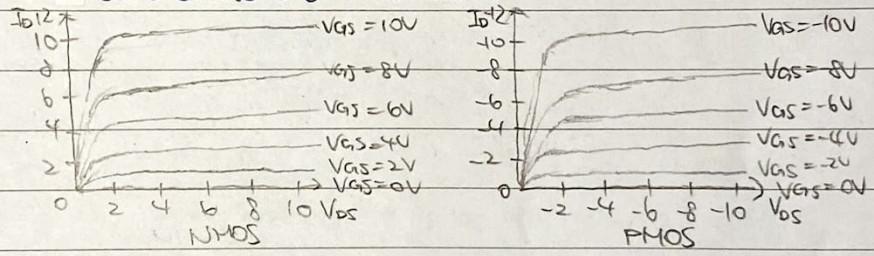


CMOS inverter

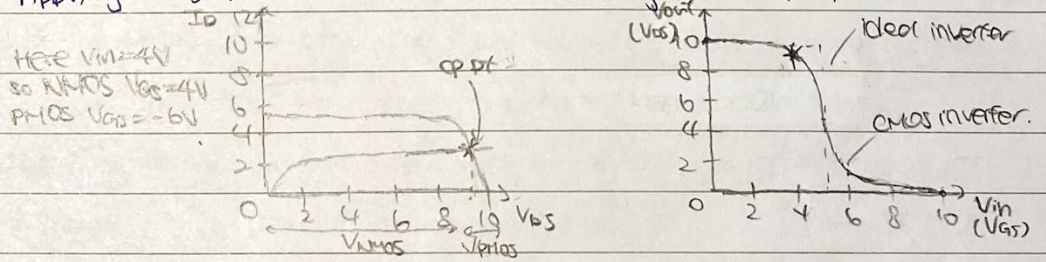
- A complementary MOS (CMOS) uses both n-type MOSFETs and p-type MOSFETs.
- p-type MOSFETs are essentially n-type MOSFETs w/ all the polarities reversed.
- The circuit of a CMOS inverter is as shown:



- The NMOS and PMOS characteristic is as shown



- Applying the graphical method to find the operating pt. → i/p-o/p characteristic.



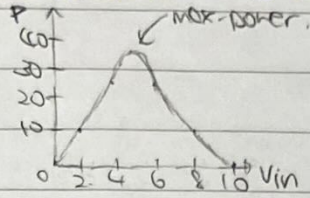
- The input-output characteristic of the CMOS inverter is much closer to the ideal inverter.
- Qualitatively, a CMOS inverter works as follows:

Vin	VGS/NMOS	VGS/PMOS	Vout
0V (LOW)	0V (OFF)	-10V (ON)	HIGH
10V (HIGH)	10V (ON)	0V (OFF)	LOW

* the NMOS and PMOS are never on simultaneously

Power consumption of CMOS inverter

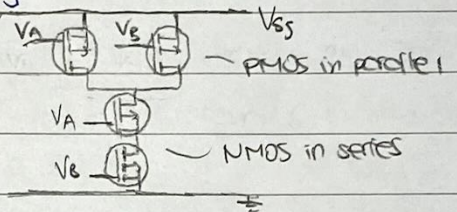
- For a CMOS inverter, the power dissipated varies w/ V_{in} as follows:



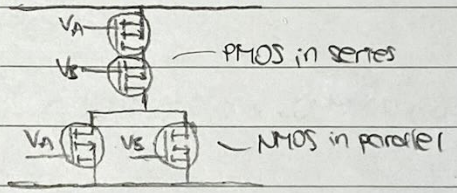
- Neither $V_{in}=10V$ or $V_{in}=0V$ dissipates power.
- current only flows when the gate is changing state → power is only dissipated as the gate switches on or off. (Inverter draws max. power at $V_{in}=5V$)
- The power can be calculated using $P=I_D V$ (both I_D, V from the operating pt.).

NAND gate and NOR gate using CMOS logic

- A NAND gate can be built using CMOS as shown below:



- A NOR gate can be built using CMOS as shown below:



Logic families

- NMOS: slowish due to stray capacitance
  High power consumption
- CMOS: propagation delay ~ 8-50 ns; max. clock speed ~12-40 MHz.
  Power consumption $<10^{-6}$ W/gate when idle, $\sim 10^{-4}$ W/gate when changing at 100kHz.
- TTL (transistor-transistor logic): Propagation delay ~ 15-40 ns; max. clock speed ~ 35-200 MHz (fast)
  use bipolar transistors Power consumption $\sim 10^{-2}$ W/gate (high power consumption)
- ECL (emitter-coupled logic): High speed
  High power consumption as current flows all the time.

Boolean algebra for logic design — rigorous, easy to adapt for larger problems

Boolean algebra.

- Commutation (switching order of inputs of gate)
  - ↳ $A+B = B+A$
  - ↳ $A \cdot B = B \cdot A$
- Association (adding inputs to the gate)
  - ↳ $(A+B)+C = A+(B+C)$
  - ↳ $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- Distribution
  - ↳ $A \cdot (B+C+\dots) = A \cdot B + A \cdot C + \dots$
  - ↳ $A+(B \cdot C \dots) = (A+B)(A+C)\dots$
- Absorption
  - ↳ $A+(A \cdot C) = A$
  - ↳ $A \cdot (A+C) = A$
- ORs
  - ↳ $A+0 = A$
  - ↳ $A+A = A$
  - ↳ $A+1 = 1$
  - ↳ $A+\bar{A} = 1$
- ANDs
  - ↳ $A \cdot 0 = 0$
  - ↳ $A \cdot A = A$
  - ↳ $A \cdot 1 = A$
  - ↳ $A \cdot \bar{A} = 0$
- AND takes over precendence over OR
- Every boolean law has a dual — any valid statement is also valid w/
  - ↳ · replaced by +
  - ↳ + replaced by ·
  - ↳ 0 replaced by 1
  - ↳ 1 replaced by 0.

e.g. $A+0=A$ ⇕ $A \cdot 1 = A$

De Morgan's theorem.

- In a simple expression ($A+B+C$ / $A \cdot B \cdot C$), we can change all the operators from OR to AND and vice versa, provided we put a bar over each term individually and a further bar over the whole expression.
  - ↳ $\overline{A+B+C+\dots} = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \dots$ or $\overline{A+B+C+\dots} = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \dots$
  - ↳ $\overline{A \cdot B \cdot C \cdot \dots} = \bar{A}+\bar{B}+\bar{C}+\dots$ or $\overline{A \cdot B \cdot C \cdot \dots} = \bar{A}+\bar{B}+\bar{C}+\dots$
- De Morgan's theorem for 2 variables can be proved by exhaustion. This can be extended to more variables by induction (grouping variables into 1 variable).

Every variable in every term

- A useful technique is to expand each term until it includes one instance of each variable (or its complement). It may be possible to cancel terms in this expanded form.
- We can expand by multiplying by $1 = B + \bar{B}$. (e.g.: $A = AB + A\bar{B}$).

- e.g.: Simplify $X \cdot Y + \bar{Y} \cdot Z + X \cdot Z + X \cdot Y \cdot Z$

$$X \cdot Y + \bar{Y} \cdot Z + X \cdot Z + X \cdot Y \cdot Z = X \cdot Y \cdot Z + X \cdot Y \cdot \bar{Z} + X \bar{Y} Z + \bar{X} \bar{Y} Z + X \cdot Y \cdot Z + X \bar{Y} \cdot Z + X \cdot Y \cdot Z$$
$$= X \cdot Y \cdot Z + X \cdot Y \cdot \bar{Z} + X \cdot \bar{Y} \cdot Z + \bar{X} \cdot \bar{Y} \cdot Z$$
$$= X \cdot Y \cdot (Z + \bar{Z}) + \bar{Y} \cdot Z (X + \bar{X})$$
$$= X \cdot Y + \bar{Y} \cdot Z$$

Expanding variables.

- Another useful technique is to expand a variable by multiplying by $1 = 1 + B$

- eg: Simplify $A + \bar{A} \cdot B$ w/o using commutation over multiplication.

$$A + \bar{A} \cdot B = A(1 + B) + \bar{A} \cdot B$$
$$= A + A \cdot B + \bar{A} \cdot B$$
$$= A + B \cdot (A + \bar{A})$$
$$= A + B.$$

Algebraic logic design

- If we design a logic circuit using gates of only certain types, we would only need to stock a limited range of gates → readily available and cheap.
- To design a logic circuit using only NAND/NOT gates, we want a sum of products for the output Y, then apply De Morgan's theorem.
  - ↳ e.g.: $Y = (\bar{A}_3 + \bar{A}_1 \cdot \bar{A}_2) \cdot B = \overbrace{\bar{A}_3 B + \bar{A}_1 \bar{A}_2 B}^{\text{sum of products } (Y)} = \overline{\overline{\bar{A}_3 B} \cdot \overline{\bar{A}_1 \bar{A}_2 B}}$
- To design a logic circuit using only NOR/NOT gates, we want a sum of products for the outputs complement $\bar{Y}$, then apply De Morgan's theorem and invert.
  - ↳ e.g.: $\bar{Y} = \overbrace{A_1 A_3 + A_2 A_3 + \bar{B}}^{\text{sum of products } (\bar{Y})} = \overline{\bar{A}_1 + \bar{A}_3} + \overline{\bar{A}_2 + \bar{A}_3} + \bar{B} \rightarrow Y = \overline{\overline{\bar{A}_1 + \bar{A}_3} + \overline{\bar{A}_2 + \bar{A}_3} + \bar{B}}$

* We could get the NOR expression by applying De Morgan's theorem on the NAND expression, but it is generally not optimised (uses more gates)

  - ↳ e.g.: $Y = \overline{\overline{A_3 \cdot \bar{B}} \cdot \overline{A_1 \bar{A}_2 \cdot \bar{B}}} = \overline{(A_3 + \bar{B}) \cdot (A_1 + A_2 + \bar{B})} = \overline{\overline{A_3 + \bar{B}} + \overline{A_1 + A_2 + \bar{B}}}$

* A NOT gate can be made from a NAND/NOR gate by connecting the inputs.

standard boolean forms.

- Sum of products (SOP): usually best to write down an expression for Y directly.
- Product of sums (POS): usually best to write down an expression for $\bar{Y}$ and use De Morgan's theorem (twice — each individual product to sum; sum of sums to product of sums).
- It is not easy to convert between SOP and POS by algebraic manipulation → best to consider all the binary permutations to find Y and $\bar{Y}$ (use karnaugh maps).

VHDL.

- Very high speed integrated circuits Hardware Description Language (VHDL) describes circuits in terms of design entities.
- Each entity consists of 2 parts — the interface and the architecture.

```
entity component_name is
    list of input and output ports
end component_name;
```
entity

```
architecture arch_name of component_name is
    declaration of internal signals;
begin
    description of what the entity does and how it's implemented
end arch_name;
```
architecture

- To use a gate entity in a larger design, we use the following syntax:

unique label to refer to this gate : entity name of gate (name of gate architecture)
port map ("port list" of signals connected to gate);

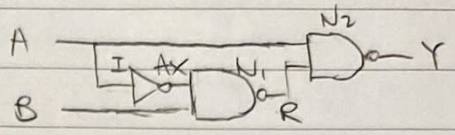
- eg: NOT gate (inverter), $Y = \bar{A}$

```
entity INV is
    port (A: in BIT;
          Y: out BIT);
end INV;
architecture IMOD of INV is
begin
    Y <= not A;
end IMOD;
```
(VHDL operator for "equals")

- eg: 2 port NAND gate, $Y = \overline{A \cdot B}$

```
entity NAND2 is
    port (A, B: in BIT;
          Y: out BIT);
end NAND2;
architecture NA2M of NAND2 is
begin
    Y <= not (A and B);
end NA2M;
```

-eg. Create an entity "ent" w/ architecture "arch" that represents the logic circuit $Y = (\overline{A \cdot B}) \cdot A$

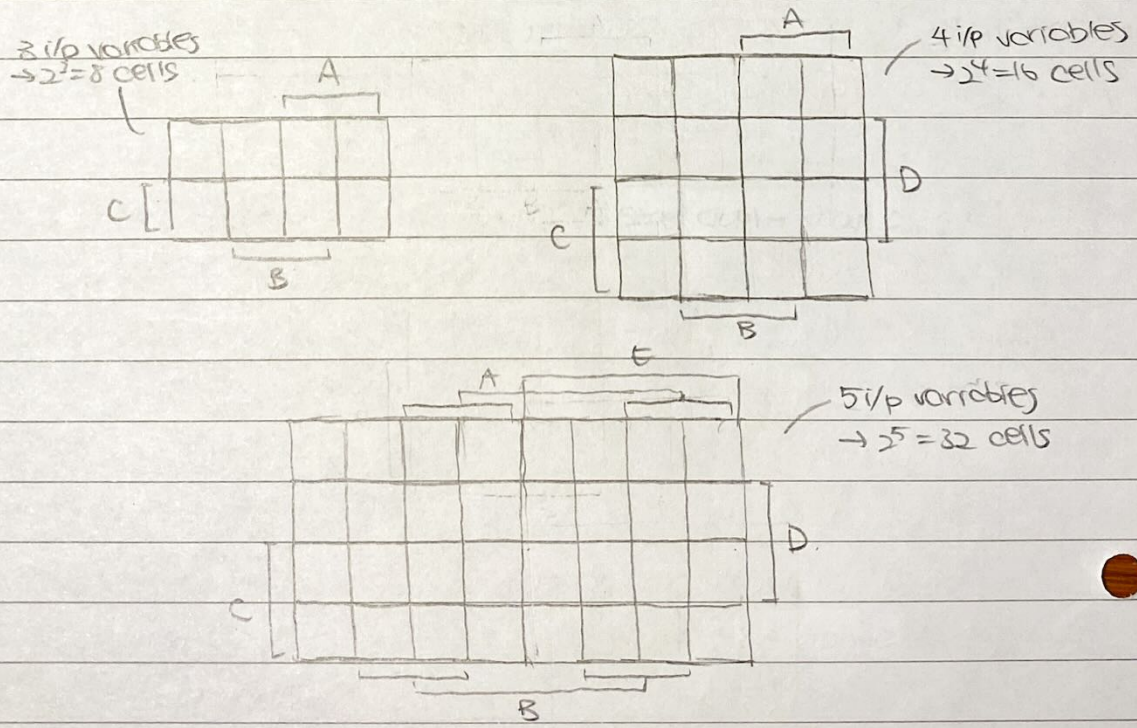



```
entity ent IS                    -- extended signals.
  port ( A, B : in BIT;
         Y    : out BIT );
end ent;
architecture arch of ent is;
  signal AX, R : BIT;            -- internal signals
begin                            -- i/p  o/p.
  I : entity INV(IMOD) port map (A, AX);
  N1 : entity NAND2 (NA2M) port map (AX, B, R);
  N2 : entity NAND (NA2M) port map (A, R, Y);
                                   -- i/p  o/p
end arch;
```

Karnaugh maps for logic design

Karnaugh maps.

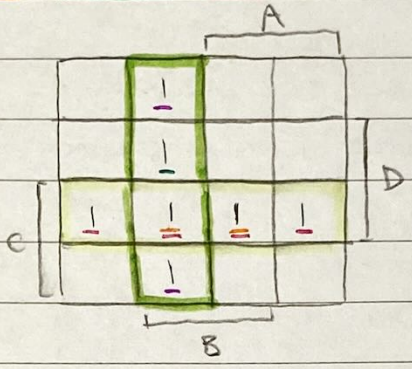
- Karnaugh maps are a powerful visual tool for carrying out simplification and manipulation of logic expressions w/ up to 5 variables.
- The Karnaugh map is a rectangular array of cells. Each possible state of the i/p variables corresponds uniquely to one of the cells, in which we write the corresponding o/p states
- Karnaugh maps for 3/4/5 variables are shown below:



Using Karnaugh maps

- After writing the output states (0 or 1) on the Karnaugh map, we try to group the 1s(Y) / 0s($\bar{Y}$) into rectangles that are as large as possible (so less are needed).
- for a 5-variable Karnaugh map, we try to group the 1s(Y)/0s($\bar{Y}$) into blocks that are as large as possible when 2 x 4-variable Karnaugh maps are overlapped.
- Karnaugh maps wrap round top-bottom and side-to-side
- Each rectangle/block corresponds to a term in the final simplified boolean expression.

- e.g.: Simplify $\bar{A}\cdot B\cdot\bar{D} + B\cdot C\cdot D + \bar{A}\cdot B\cdot\bar{C}\cdot D + C\cdot D$ using a Karnaugh map.



∴ The simplified expression is $\bar{A}\cdot B + C\cdot D$

Don't care states

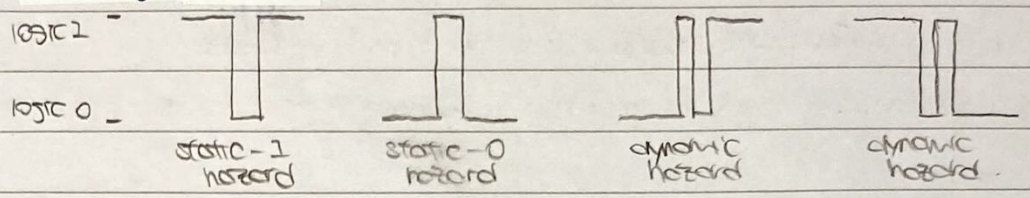
- In some applications, the output state for certain combinations of input variables may not matter — such states are don't care states and are denoted by X.
- The don't care states can be chosen to be 0 or 1, whichever helps to produce the simplest logic (i.e. fewest simplest terms).

Karnaugh map circuit design.

- To design a logic circuit using only NAND/NOT gates,
  - ↳ Write down simplest sum-of-products for the output Y from the Karnaugh map. (1s).
  - ↳ Apply De Morgan's theorem ($+ \rightarrow \cdot$)
- To design a logic circuit using only NOR/NOT gates,
  - ↳ Write down simplest sum-of-products for the output's complement $\bar{Y}$ from the Karnaugh map (0s)
  - ↳ Apply De Morgan's theorem ($\cdot \rightarrow +$)
- Sometimes a simpler logic circuit can be produced by mapping the opposite way (group 0s for NAND; group 1s for NOR) and use a final inverter gate to change the output back again

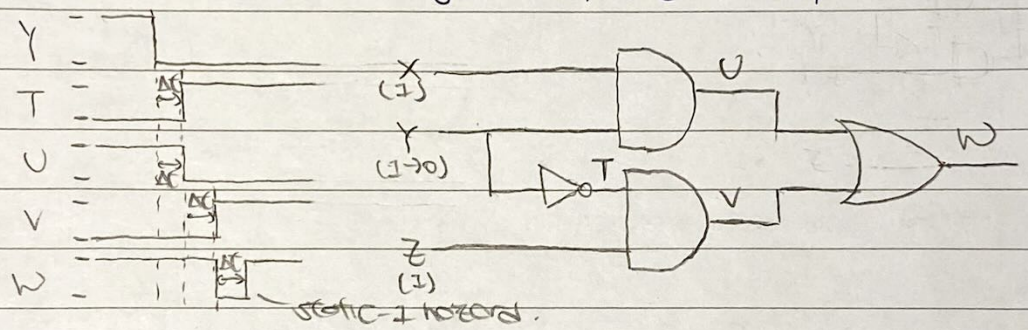
Hazards

- A static hazard is when a signal undergoes a momentary transition when it is supposed to remain unchanged
- A dynamic hazard is when a signal changes more than once when it is supposed to change just once.



- Hazards arise due to propagation delay of gates.

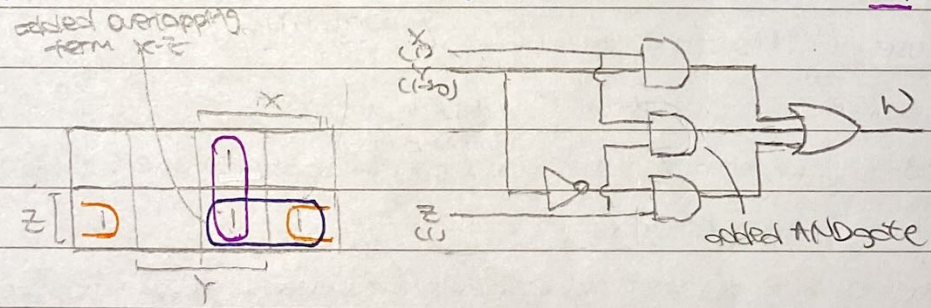
- eg: Consider the output when Z=1, X=1 and Y changes from 1 to 0. for the circuit shown. All gates have propagation delay Δt



Removing hazards

- Fix a static-1 hazard by drawing the karnaugh map of the output Y. Make sure all the sum-of-product terms overlap.
- Fix a static-0 hazard by drawing the karnaugh map of the outputs complement $\bar{Y}$. Make sure all the sum-of-product terms overlap.
- Fix dynamic hazards by redesigning the circuit to simplify the logic (beyond the scope of this course)

- eg: Remove the static-1 hazard from the circuit above ($W = X \cdot Y + \bar{Y} \cdot Z$).



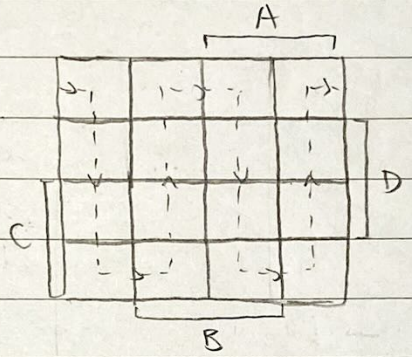
The hazard is removed from the circuit by adding on extra AND gate so now

$W = X \cdot Y + \bar{Y} \cdot Z + X \cdot Z$

Unit distance codes

- Applications exist where a code is req. in which not more than 1 bit changes between consecutive nos. (eg shaft encoder).
- In some designs, there may be unwanted transitory states due to poor positioning, which can be avoided using unit distance codes.
- As we move from a cell to its neighbour in a karnaugh map, only 1 variable changes (since the map wraps round, this is also true moving from top/bottom or left/right).
- This property of karnaugh maps can be used to produce a code in which only 1 bit changes at a time → unit distance code.

-eg: Grey code -



A	B	C	D
0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0

A	B	C	D
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

* other unit distance codes are possible using a different path around the karnaugh map.

Sequential Logic

Binary numbers

Binary numbers

$2 \mid 21$
$2 \mid 10 \ldots 1$
$2 \mid 5 \ldots 0$
$2 \mid 2 \ldots 1$
$1 \ldots 0$

$21_{10} = 10101_2$

- Binary is base 2. Each digit is either 0 or 1.
- We can convert decimal into binary using the remainders from successive division.
- A binary digit is called a bit; An 8 bit storage location is a byte.

Octal numbers

$8 \mid 42$
$5 \ldots 2$

$42_{10} = 52_8$

- Octal is base 8. Each digit is either 0, 1, 2, 3, 4, 5, 6, 7 or 8.
- We can convert decimal into octal using the remainders from successive division.
- We can convert between binary and octal by dividing the binary digits into 3-bit groups.

Hexdecimal numbers

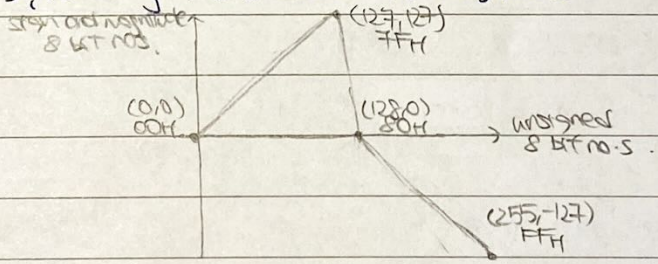
$16 \mid 63$
$3 \ldots 15$

$63_{10} = 3F_H$

- Hexdecimal is base 16. Each digit is either 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E or F.
- We can convert decimal into hexdecimal using the remainder from successive division.
- We can convert between binary and hexdecimal by dividing the binary digit into 4-bit groups.
- Two hexdecimal digits are often used to specify the contents of a byte.
- In assembler programming, hexdecimal nos are indicated using $ (e.g. $57 means 57_H)

Negative numbers – sign and magnitude representation

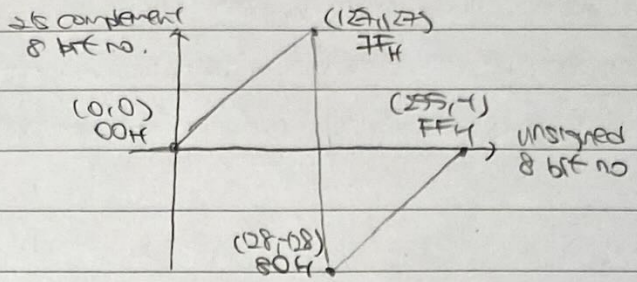
- For an 8-bit no. using the sign and magnitude representation, the MSB indicates the sign (0: +, 1: −), remaining 7 bits contain the magnitude.



- The sign and magnitude representation is not used because:
  - ↳ We have two zeros: +0 (00_H) and −0 (80_H) → we have 1 less no.
  - ↳ We req. separate circuits to add +ve and −ve nos.

Negative numbers — 2's complement

- To convert between +ve and -ve 2's complement no., we invert all the bits and add 1.



- When we do this to an 8-bit binary no. x,
  - ↳ Invert all the bits : $x \rightarrow (255 - x)$
  - ↳ Add 1 : $(255 - x) \rightarrow (256 - x)$ $\quad * 256_{10} = 100_H$
- Since 100_H will not fit into an 8-bit byte, it resets 00_H and sets the carry flag
  If we choose to ignore the carry flag, then 256-x will behave just like 00_H. (ignore 7th bit.)
- Therefore, we can use normal binary arithmetic to manipulate the 2's complement and it behaves just like -x.
- When working w/ unsigned nos, we can detect overflow using the carry flag. As we deliberately ignore the carry flag when working w 2's complement nos, we detect overflow by checking if carry into the MSB ≠ carry out of the MSB instead.

-e.g: 15 + 15

	0	0	0	0	1	1	1	1 ($+15_{10}$)
	0	0	0	0	1	1	1	1 ($+15_{10}$)
(0)	0	0	0	1	1	1	1	0

(0) ← 0 ← 0 (× ×)

no carry out of MSB ; no carry into MSB
→ no overflow

-e.g: -127 + (-2)

	1	0	0	0	0	0	0	1 (-127_{10})
	1	1	1	1	1	1	1	0 (-2_{10})
(1)	0	1	1	1	1	1	1	1

(1) ← 0 ← 1 (✓ ×)

carry out of MSB ; no carry into MSB
→ overflow

Binary coded decimal (BCD)

- For a binary coded decimal no, each decimal bit is coded as a 4 bit word. (a group of bits)
  - ↳ eg: $1248_{10} = 0001\ 0010\ 0100\ 1000_{BCD}$
  - ↳ eg: $1234_{10} = 0001\ 0010\ 0011\ 0100_{BCD}$
- BCD is not an efficient way of storing the nos, but it is easy to code and decode.
- This is 8421 BCD because the bits have these weightings.
- Other variants of BCD include
  - ↳ 2421 BCD — the top bit only has a weight of 2
  - ↳ Excess-3 code — record each digit value plus 3 using the 8421 weights.

Alphanumeric character codes

- In the standard version of ASCII (American standard Code for Information Interchange), 7 bits are used and the remaining bit is set to zero or used for parity.
- The first 32 numbers are control codes originally used for controlling modems.
- The rest are upper case letters, lower case letters, numbers and punctuation.
- An extended version of ASCII uses all 8 bits to provide 128 additional graphics characters which vary from computer to computer.
- Other alphanumeric code systems include EBCDIC (on IBM systems) and Unicode (a 16-bit system that inc. chinese characters etc..)

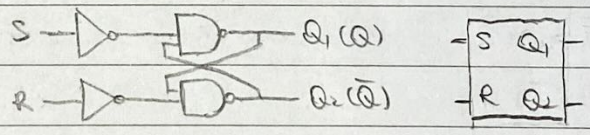
Parity.

- Parity is the simplest error detection code used for transferring binary data through an imperfect data channel
- In the transmission circuitry, it involves setting a particular bit in each binary no. so as to ensure that an even no. of bits are always on.
- In the receiver circuitry, we count the no. of bits are on in each no. and reject any that have an odd no.
- Parity can only detect 1-bit errors — more serious errors may go undetected.

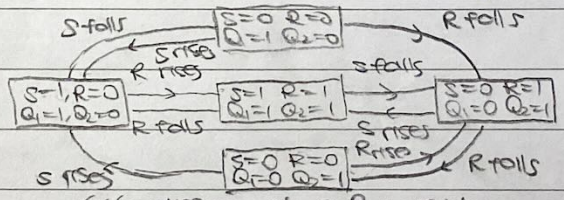
BISTABLES.

Set-Reset bistable (SR bistable)

- The circuit of a SR bistable is as shown:



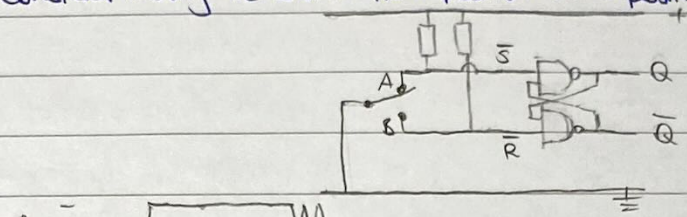
- S will turn Q_1 ON (set) and Q_2 OFF. They will stay at this state until R turns on
- R will turn Q_1 OFF (reset) and Q_2 ON. They will stay at this state until S turns on.
- The circuit state depends not only on the current values of S and R, but also on their values in the past — i.e. we have memory.
- If we do not allow S=R=1, then we can say $Q_2 = \bar{Q}_1$, or denote Q_1, Q_2 as $Q, \bar{Q}$.
- This state is forbidden as we have a race and therefore we cannot predict what the next state will be.

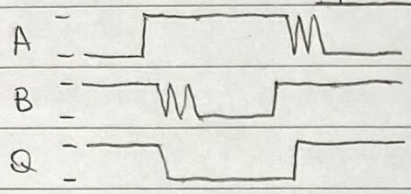


state diagram of SR bistable.

contact debouncing.

- An S-R bistable could be used to prevent glitches due to contact debouncing.
- Consider moving the switch from position A to position B and back to position A.



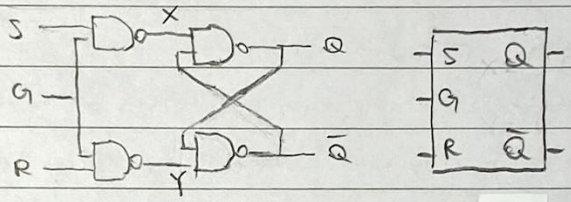


Note here $A = \overline{S}$, $B = \overline{R}$

(Both A and B are active LOW signals).

Gated S-R bistable.

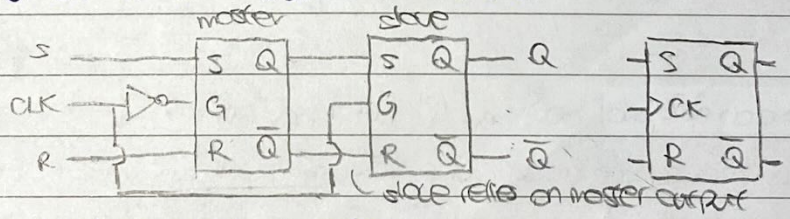
- The circuit of a gated S-R bistable is as shown:



- To set Q, $S\overline{G}$ needs to be LOW, i.e. both S and G are HIGH; To reset Q, $R\overline{G}$ needs to be LOW, i.e both R and G are HIGH.
- This means the bistable can only change states when the gate input G is HIGH
- The gate G is often connected to a clock signal – a regular signal w/ repetitive timing, which allows for accurate timing and can avoid hazards / races.

Master-slave bistable

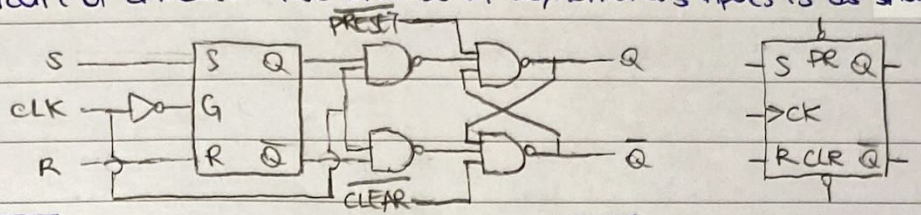
- The circuit of a master-slave bistable is as shown:



- When CLK is LOW, the master bistable is sensitive to input signals.
- When CLK rises (HIGH), the slave bistable is sensitive to input signals, which are the outputs of the master bistable. (State of master bistable transferred to slave bistable)
- The overall output only changes when the CLK input rises.

Master-slave bistable with asynchronous inputs

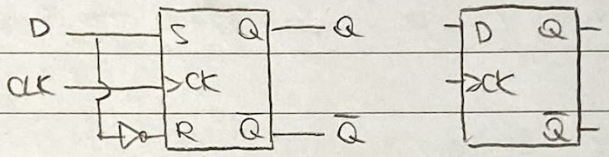
- The circuit of a master-slave bistable w/ asynchronous inputs is as shown:



- If $\overline{\text{PRESET}}$ goes to LOW, Q goes to HIGH; If $\overline{\text{CLEAR}}$ goes to LOW, Q goes to LOW.
  Both $\overline{\text{PRESET}}$ and $\overline{\text{CLEAR}}$ are active LOW signals.
- S, R are both gated, connected to a CLK signal → synchronous inputs.
- $\overline{\text{PRESET}}$ and $\overline{\text{CLEAR}}$ are not gated, so not connected to a CLK signal → asynchronous inputs.

D-type latch

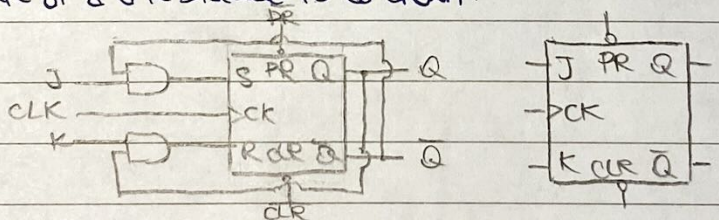
- The circuit of a D-type latch is as shown:



- A D-type latch has only 1 input
- The input is clocked through the bistable as CLK goes HEIGH.
- The D-type latch is useful for the retrieval of signals — used in shift registers.

J-K bistable.

- The circuit of a JK bistable is as shown:



- The addition of the AND gates avoids the forbidden state S=1, R=1.
- The following are the r/p-o/p table and the characteristic table of the JK bistable.

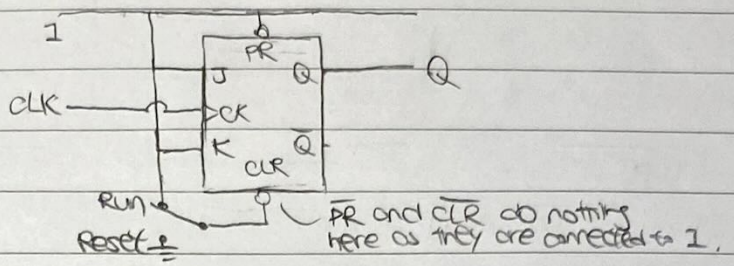
	inputs when CLK is LOW		outputs when CLK rises	output before → after		required inputs.	
	Input J	Input K	Q(n+1)	Q(n)	Q(n+1)	Input J	Input K
unchanged	0	0	Q(n)	0	0	0	x
set	0	1	0	0	1	1	x
reset	1	0	1	1	0	x	1
toggle	1	1	$\overline{Q(n)}$	1	1	x	0

- When J=1, K=1, the JK bistable toggle the output
- If at any time when CLK is LOW and a signal exists at the input encouraging the bistable to change state, that change will take place only when CLK rises.

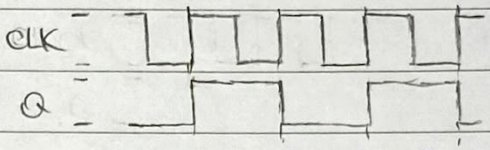
Application of bistables

Divide by 2 counter

- The circuit of a divide by 2 counter is as shown:



- The timing diagram is as follows:

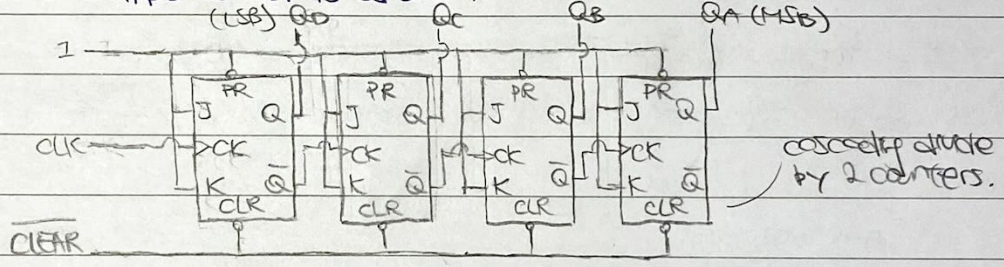


- J=1, K=1 → output toggles at the rising edge of CLK
- Period of Q is doubled / frequency of Q is halved w.r.t. CLK (Divide by 2)

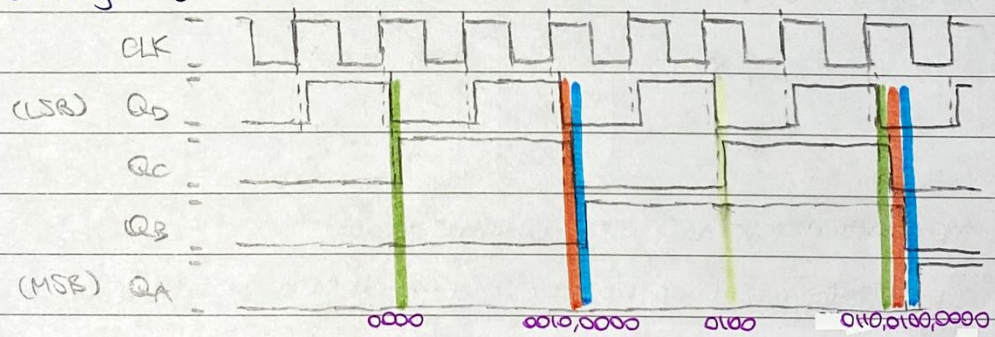
* Propogation delay is not being considered in the timing diagram

Ripple counter

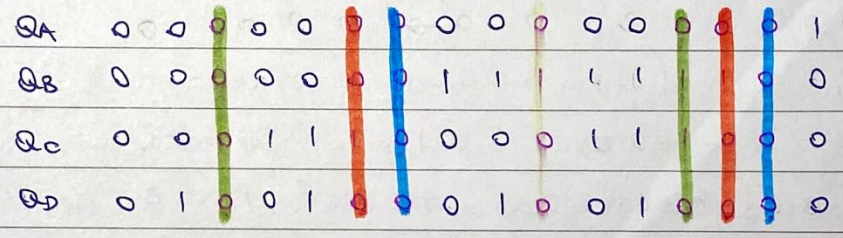
- The circuit of a ripple counter is as shown:



- The timing diagram is as follows:

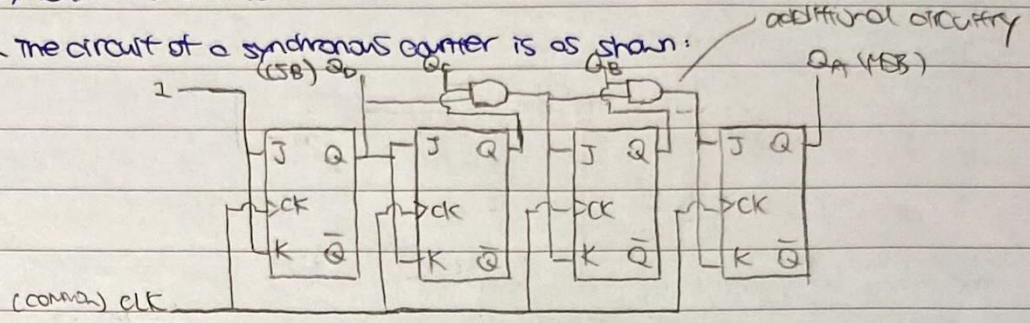


- We can use the ripple counter to count up in binary (from 0000 to 1111).
- The propogation delays between CLK signals changing and the J-K output changing results in unwanted transitory states. (glitches)

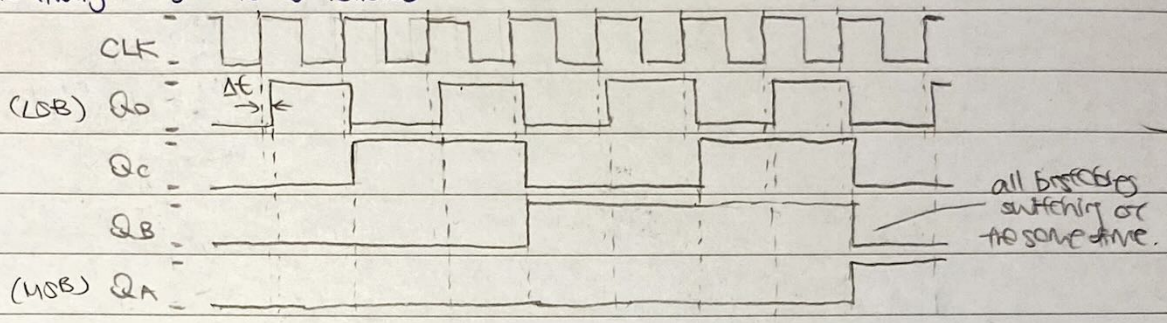


Synchronous counter

- The circuit of a synchronous counter is as shown:



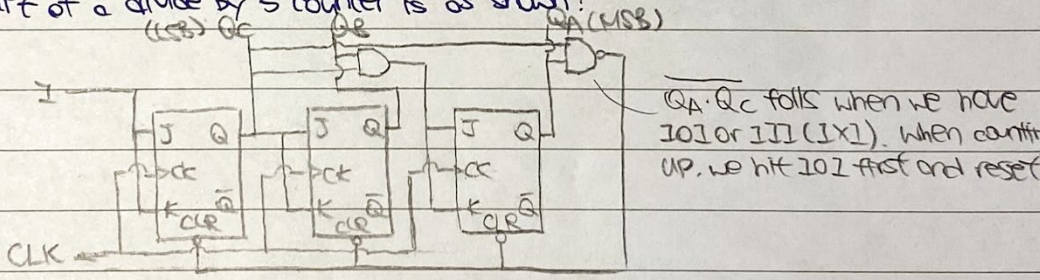
- The timing diagram is as follows



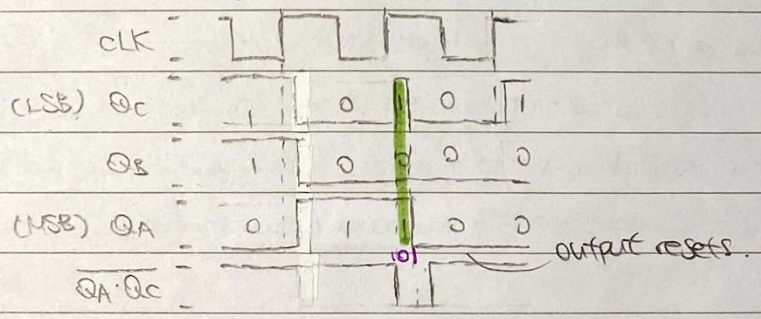
- By using a common CLK and adding additional circuitry, we can create a synchronous counter that does not have any transitory states
- Although there is propagation delay between CLK and Q_i, the delays are not accumulated across Q_i.

Divide by 5 counter

- The circuit of a divide by 5 counter is as shown:



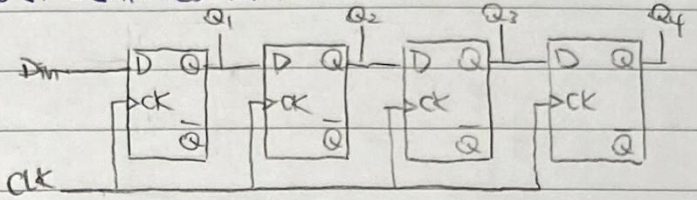
- The timing diagram is as follows:



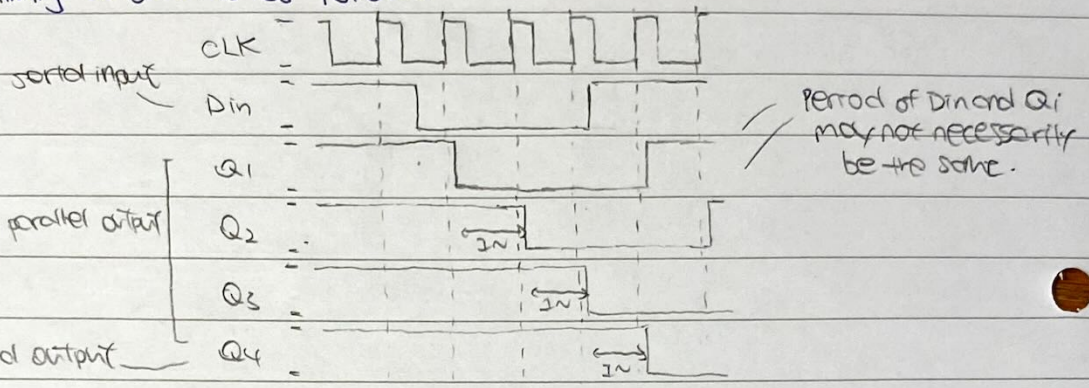
- Asynchronous clear can be used to produce divide by N counters for N not equal to a power of 2.
- However glitches arise because of the use of asynchronous inputs.
- By adding additional circuitry, we can make a synchronous design w/o such glitches.

shift register

- The circuit of a shift register is as shown:



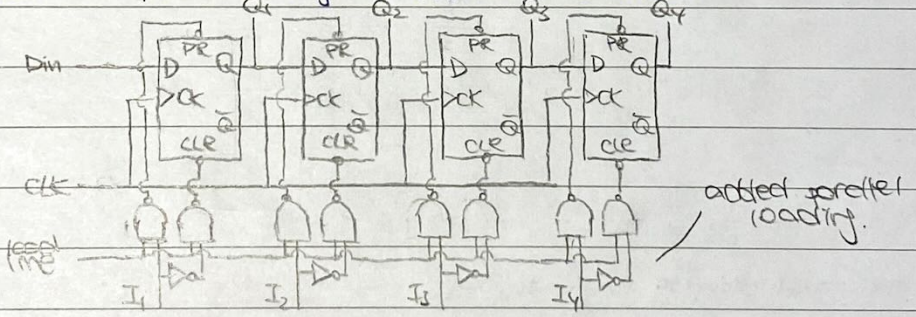
- The timing diagram is as follows:



- The inputs of each D-type latch are retimed and shifted by 1 CLK period rel. to the previous D-type latch
- Therefore the output datastream is the same as the input data but in this case, the output is retimed and delayed by roughly 4 CLK periods.

parallel loading shift register.

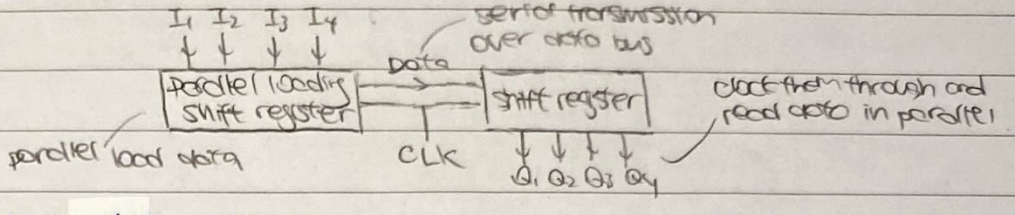
- The circuit of a parallel loading shift register is as shown:



- When the load line is HIGH for a short period, I_i is loaded to Q_i.
- This could be then clocked through to Q_4 for a serial output (of $I_4 I_3 I_2 I_1$). time →
- The parallel loading shift register allows us to read a parallel datastream and output a serial datastream at Q_4 (or output data in parallel at Q_i).

Serial data link

- A serial data link is connected as shown:

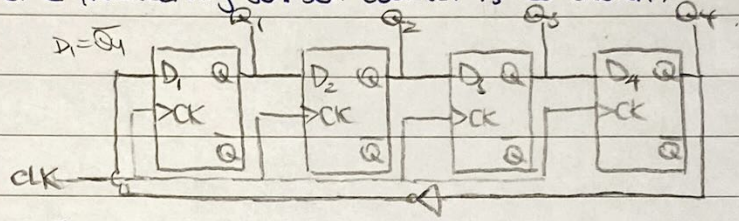


- A parallel loading shift register is used to send one data bit at a time across a serial link.
- Doing so would req. less wires than would be req. for a fully parallel link.

Twisted ring Johnson counter

Johnson counter we have $D_1 = \overline{Q_4}$

- The circuit of a twisted ring Johnson counter is as shown:



- A twisted ring Johnson counter uses a shift register of length N to produce a sequence of length of 2N. (Assume initial state of 0000).
- Several other sequences are possible, depending on the initial state.
- For ring counters in general ($D_1 = fn(Q_1, Q_2, Q_3, Q_4)$), the sum of no. of states in each sequence is 2^N

Synchronous logic design

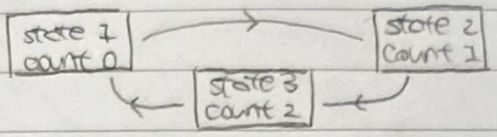
Synchronous logic design.

- By only using synchronous inputs, we can avoid glitches (unwanted transitory states).
- There are 5 stages for synchronous logic design:
  - ↳ 1) State diagram: Define o/p in each state, define transitions for all combination of i/ps.
  - ↳ 2) Bistable allocation: n bistables provided 2^n states. Assign code to each state and note the unused states
  - ↳ 3) State transition table: Work out bistable i/ps req. to perform correct state transitions. Handle every combination of current state and input variables (inc. unused states)
  - ↳ 4) Karnaugh maps: Work out logic to implement function req. by the state transition table.
  - ↳ 5) Circuit: All the bistables should be clocked synchronously. Translate bistable o/ps to the circuit o/ps (if necessary)

* Very important to get the state diagram right → double check.

-eg : Divide by 8 counter

① state diagram



② Bistable allocation.

- 2 bistables → $2^2 = 4$ states (3 allocated states + 1 unallocated state)

		Bistable outputs Q_A	Q_B
state 1	count = 0	0	0
state 2	count = 1	0	1
state 3	count = 2	1	0
	UNUSED	1	1

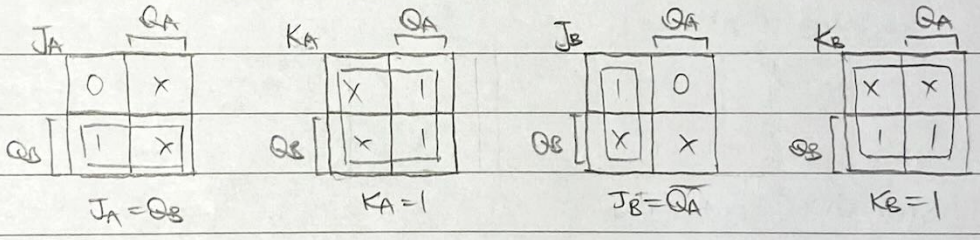
③ State transition table

	current state Q_A	Q_B	Next state Q_A	Q_B	Required inputs J_A	K_A	J_B	K_B
count = 0	0	0	0	1	0	x	1	x
count = 1	0	1	1	0	1	x	x	1
count = 2	1	0	0	0	x	1	0	x
unallocated	1	1	0	0	x	1	x	1

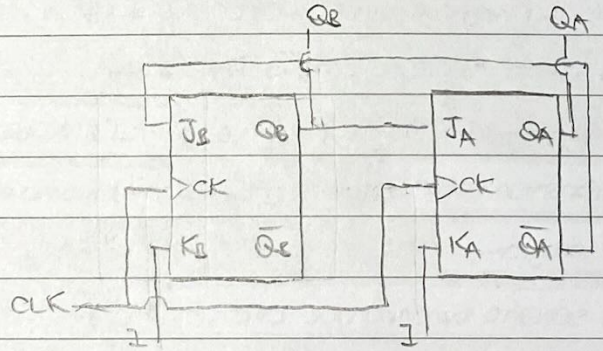
refer to the JK characteristic in the databook.

chosen to go back to 00 so it won't get stuck.

④ Karnaugh maps



⑤ circuit



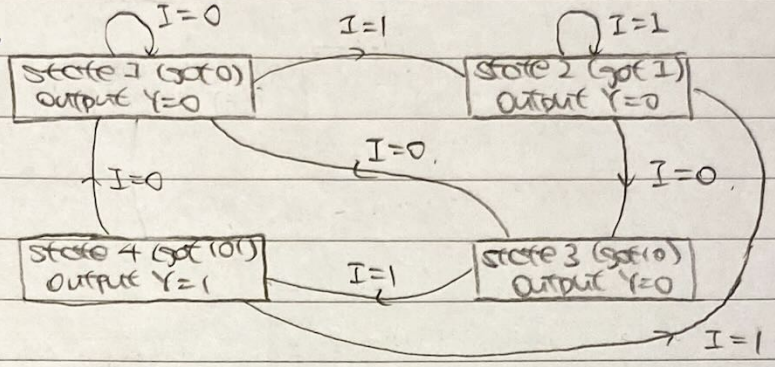
* output logic step skipped as the bistable outputs are the same as the circuit output

- e.g. sequence detector

Detect the received sequence 101 on an input I. When the sequence is detected, Y=1.

only respond to non-overlapping sequences. (10101 should give Y=1 after bit 3 but not 5)

① State diagram.



* 2 possible combination of inputs → 2 paths leaving every state.

② Bistable allocation

- 2 bistables → $2^2 = 4$ states (4 allocated states + 0 unallocated states)

	Bistable outputs Q_A	Q_B
state 1 got 0	0	0
state 2 got 1	0	1
state 3 got 10	1	0
state 4 got 101	1	1

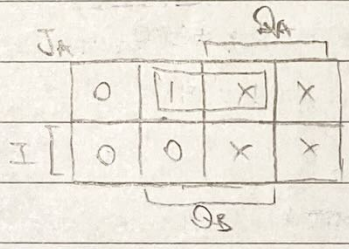
③ State transition table

determine next state using state diagram

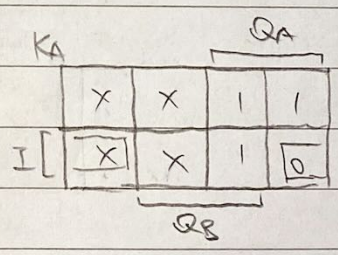
	Input I	Current state Q_A	Q_B	Next state Q_A	Q_B	Required inputs J_A	K_A	J_B	K_B
St1	0	0	0	0	0	0	X	0	X
	1	0	0	0	1	0	X	1	X
St2	0	0	1	1	0	1	X	X	1
	1	0	1	0	1	0	X	X	0
St3	0	1	0	0	0	X	1	0	X
	1	1	0	1	1	X	0	1	X
St4	0	1	1	0	0	X	1	X	1
	1	1	1	0	1	X	1	X	0

refer to the JK characteristic in the datasheet.

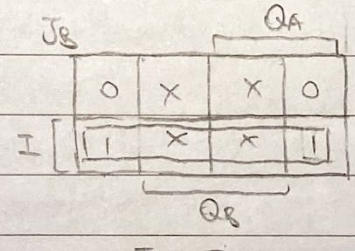
④ Karnaugh maps



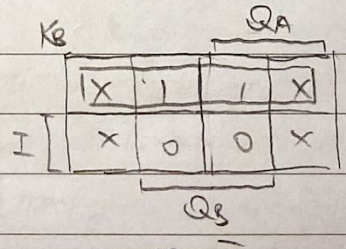
$J_A = Q_B \cdot \overline{I}$



$\overline{K_A} = \overline{Q_B} \cdot I \Rightarrow K_A = \overline{\overline{Q_B} \cdot I}$



$J_B = I$



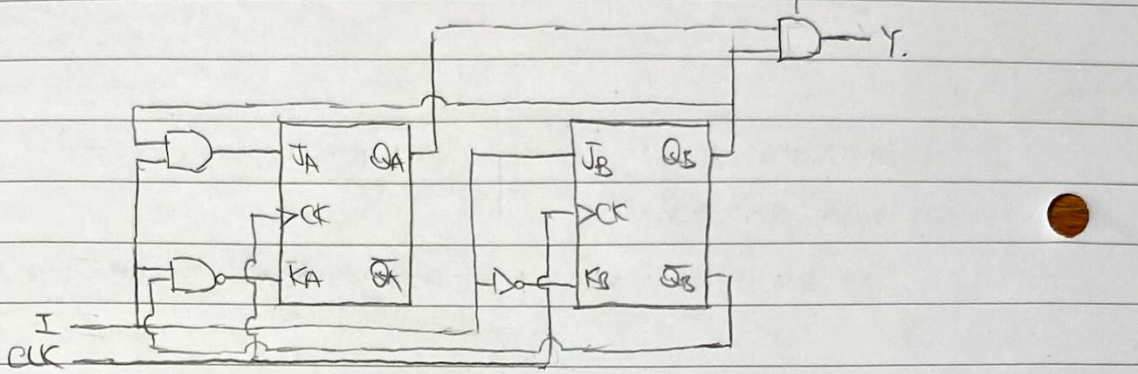
$K_B = \overline{I}$

⑤ circuit.

		Bistable outputs QA QB	Circuit output Y
state 1	set 0	0 0	0
state 2	set 1	0 1	0
state 3	set 10	1 0	0
state 4	set 101	1 1	1

so $Y = A \cdot B$.

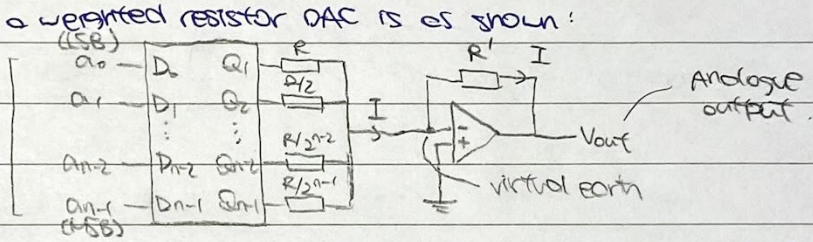
translate bistable o/p to circuit o/p.



Analogue conversion and adder circuits.

weighted resistor DAC — digital to analogue converter

- The circuit of a weighted resistor DAC is as shown:



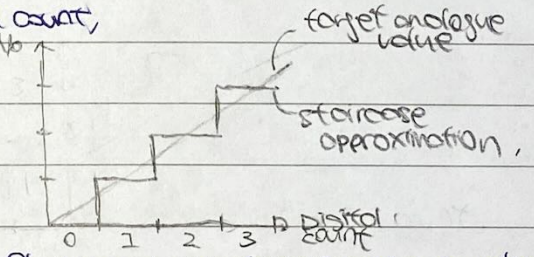
- Assume the op-amp is ideal → inverting input is virtual earth and $V_{out} = -IR'$

Let the latch output voltage be A for each Q=1, and 0 for each Q=0.

$$\text{KCL}: \quad I = \frac{a_0 A}{R} + \frac{2a_1 A}{R} + \ldots + \frac{2^{n-1} a_{n-1} A}{R}$$

$$\therefore V_{out} = -\frac{AR'}{R} \sum_{m=0}^{n-1} a_m 2^m$$

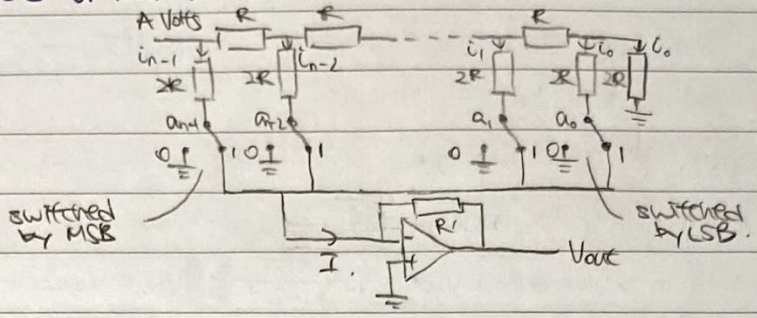
- Plotting $-V_{out}$ against the digital count,



- The staircase approximation becomes closer to the target analogue value as we increase n / size of the data word (Latch o/p voltage divided into smaller units).
- The weighted resistor DAC has a large range of resistance value → hard to implement, made from different materials so the output may drift as it warms up.
- If the tolerance of the LSB resistor is a%, then the tolerance of the MSB resistor must be $\frac{a}{2^{n-1}}$% → time consuming and expensive to achieve.

R-2R ladder DAC

- The circuit of a R-2R ladder DAC is as shown:



- Assume the opamp is ideal → inverting input is virtual earth and $V_{out} = -IR'$

In this configuration, we find that $i_{m+1} = i_m$ so $i_n = 2^n i_0$

Consider the left-most 2R resistor, w/ current i_{n-1},

$$A = i_{n-1} \cdot 2R = 2^{n-1} i_0 \cdot 2R \rightarrow i_0 = \frac{A}{2^n R}.$$

$$\text{KCL:} \quad I = \sum_{m=1}^{n-1} a_m i_m$$

$$= \sum_{m=1}^{n-1} a_m 2^m \left(\frac{A}{2^n R}\right)$$

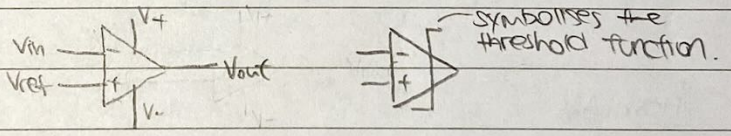
$$= \frac{A}{2^n R} \sum_{m=1}^{n-1} a_m 2^m$$

$$\therefore V_{out} = -\frac{AR'}{2^n R} \sum_{m=1}^{n-1} a_m 2^m$$

- This circuit provides similar functionality to the weighted resistor DAC but only has 2 resistor values (R and 2R).
- Therefore it is more accurate, scalable and potentially cheaper.

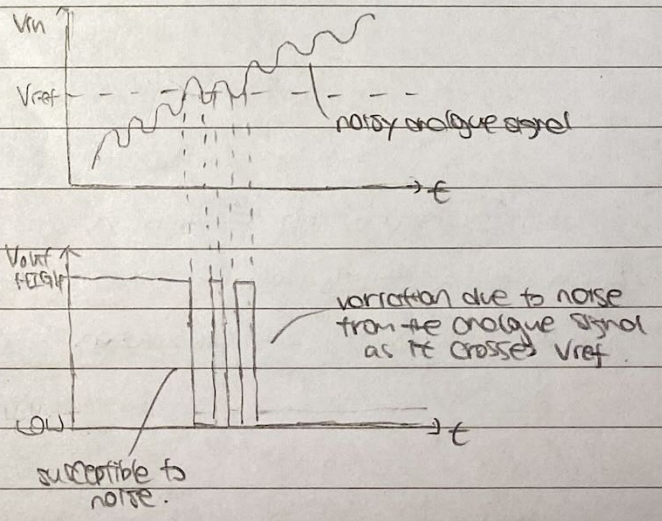
1 bit ADC (analogue to digital converter) without hysteresis

- The circuit of a 1 bit ADC (comparator) w/o hysteresis is as shown:



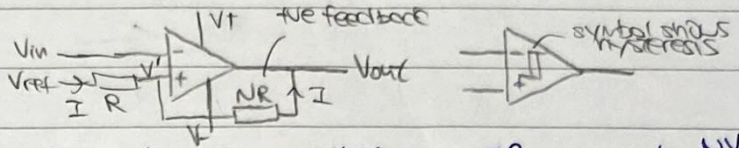
- In general, $V_{out} = \text{Gain} \times (V_{ref} - V_{in})$. However, since gain is large,

$V_{in} > V_{ref} \rightarrow V_{out} = V_-$; $V_{in} < V_{ref} \rightarrow V_{out} = V_+$



1 bit ADC with hysteresis (schmitt trigger)

- The circuit of a 1 bit ADC (comparator) w/ hysteresis is as shown:



- In general, $V_{ref} - V' = IR$; $V' - V_{out} = NIR$ $\rightarrow$ $V' = \frac{NV_{ref} + V_{out}}{N+1}$

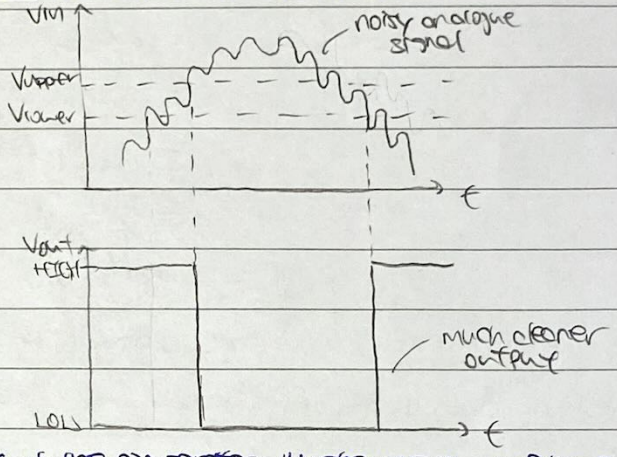
However, since gain is large, when,

$V_{in} > V' \rightarrow V_{out} = V_-$ $\rightarrow$ $V' = \frac{NV_{ref} + V_-}{N+1}$ which gives V_{lower}

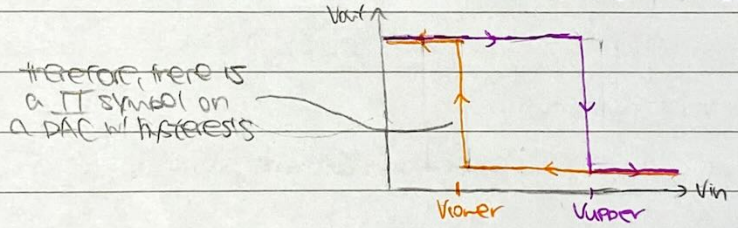
$V_{in} < V' \rightarrow V_{out} = V_+$ $\rightarrow$ $V' = \frac{NV_{ref} + V_+}{N+1}$ which gives V_{upper}.

- If Vin is initially LOW, Vin must rise above V_{upper} for Vout to be LOW.

If Vin is initially HIGH, Vin must drop below V_{lower} for Vout to be HIGH

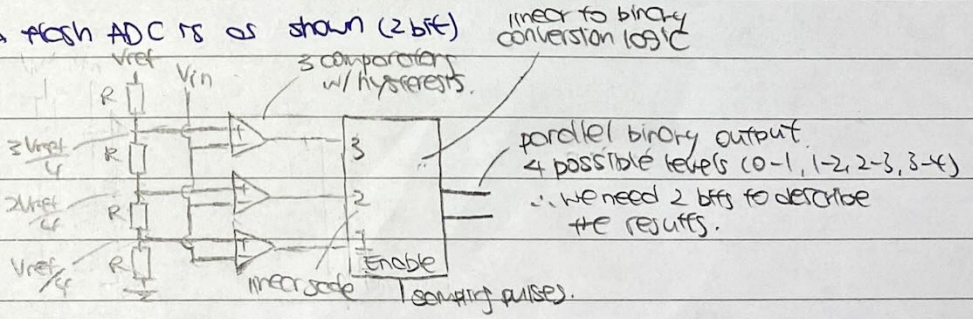


- The transfer characteristics of the comparator w/ hysteresis is as follows:



Flash ADC

- The circuit of a flash ADC is as shown (2 bit)



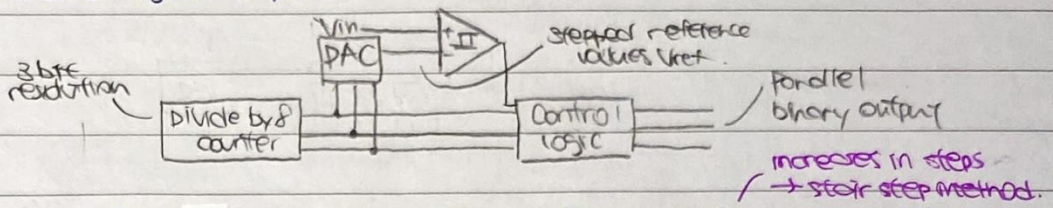
- The flash ADC is very fast — the binary word is updated at each CLK pulse.

- For a large no. of bits (higher resolution), we will need a large no. of comparators.

A n bit flash ADC req. $2^n - 1$ comparators.

Digital ramp ADC

- The circuit of a digital ramp ADC is as shown:



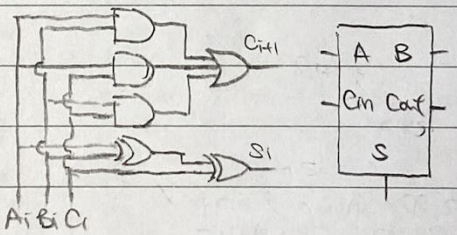
- The digital ramp ADC uses a counter + DAC to create a gradually increasing reference voltage Vref, which is used to compare w the analogue input Vin using a comparator.
- This is much slower than the flash ADC

Full adder

- For a full adder, the input-output table is as follows:

A_i	B_i	C_i	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

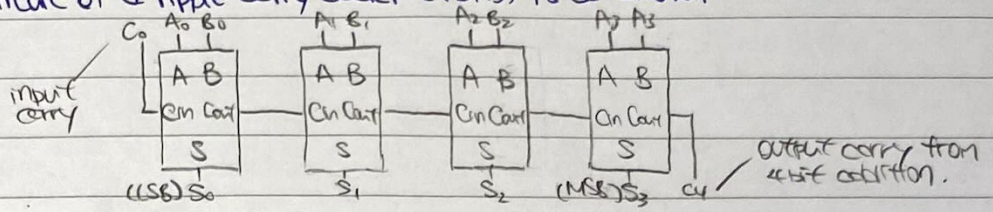
- Carry (C_{i+1}) : 2 or more inputs need to be 1 if we want $C_{i+1} = 1$
  - ↳ $C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$
- Sum (S_i) : Needs an odd no. of inputs to be 1 if we want $S_i = 1$
  - ↳ $S_i = A_i \oplus B_i \oplus C_i$
- The circuit of a full adder is as follows:



- The full adder circuit is used in the arithmetic and logic units (ALU) of microprocessors.
- There are no 3 i/p XOR gates → we use 2x 2 i/p XOR gates.

Ripple carry adder circuit

- The circuit of a ripple carry adder (4 bits) is as shown:



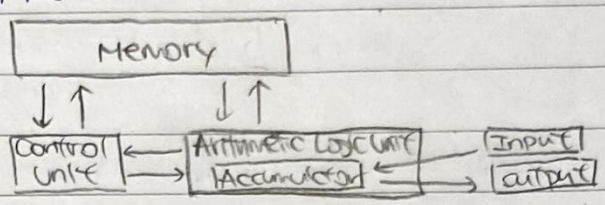
- We can cascade n full adder circuits to make an adder of n bit words
- If we input $\bar{A}_0, \bar{A}_1, \bar{A}_2$ and $\bar{A}_3$ instead, and set C_0 to 1, we would effectively add the 2's complement of A, i.e. subtract A.

Microprocessors

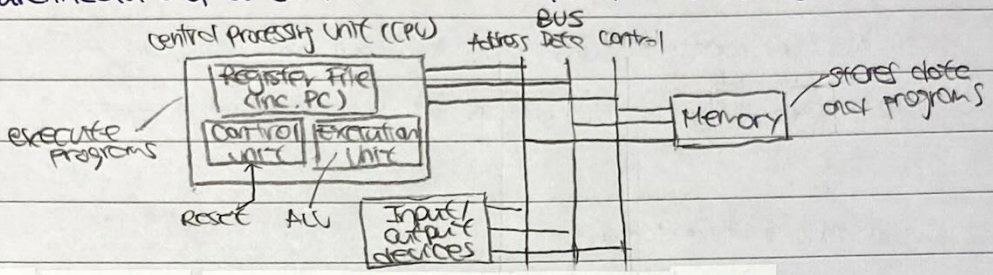
Microprocessor architectures

Microprocessor architecture

- The basic Von Neumann architecture is as follows



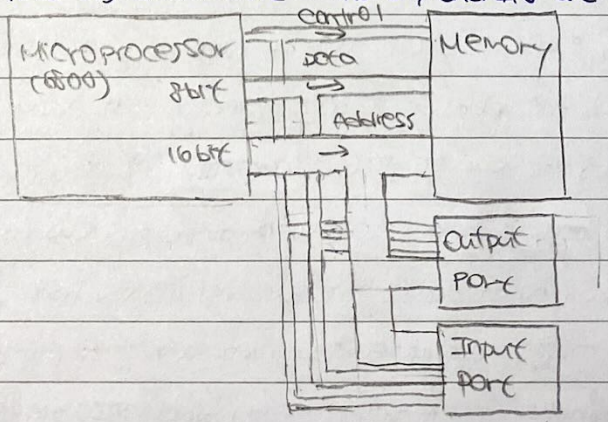
- The design was the first ever stored program machine.
- The architecture of a (simple) modern computer is as follows



- The central processing unit (CPU) / microprocessor consists of
  - ↳ An Arithmetic Logic Unit (ALU) which performs mathematical and logic operations.
  - ↳ A control unit that manages the execution of instructions
  - ↳ Multiple registers that are used for temporary storage for instructions and data.
- The memory stores instructions and data
- Input and output ports act as an interface w/ the outside world (keyboard, mouse..)
- Buses connect everything together.

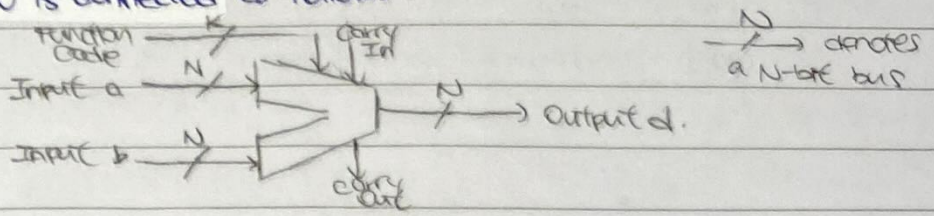
Basic bus structure.

- The data bus is bidirectional ; the address and control buses are unidirectional.
- The 6800 microprocessor has an 8 bit data bus and a 16 bit address bus. More advanced processors have more data and address lines (usually 16, 32, 64)
- The PIC does not have this architecture — memory and I/O are via registers.



Arithmetic logic unit (ALU)

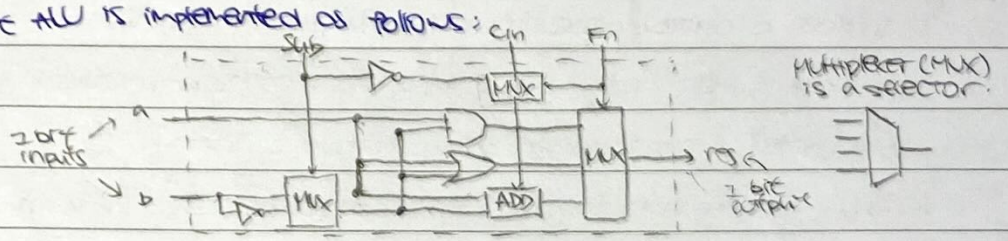
- A N-bit ALU is connected as follows:



- The ALU inputs from a register/memory location and outputs to a register/memory location.
- The ALU can perform simple 2 operand functions (e.g. a+b, a-b, a AND b, a OR b etc) where a,b could be single bits or words.
- The ALU can perform simple comparison operations (test for zero, ve etc.)
- The function to be carried out is set by the function code.

1-bit ALU implementation

- A 1-bit ALU is implemented as follows:



- This 1-bit ALU can perform 8 possible functions — a AND b, a AND $\bar{b}$, a OR b, a OR $\bar{b}$, a+b, a+b w/ carry, a-b, a-b w/ borrow.
- The function is selected by multiplexers.
- To make a N-bit ALU, we connect N× 1-bit ALUs together, using Cin, Cout.

Registers.

- Registers are used as temporary storage for instructions and data within the microprocessor.
- In the PIC12F675, there are 2 main types of registers.

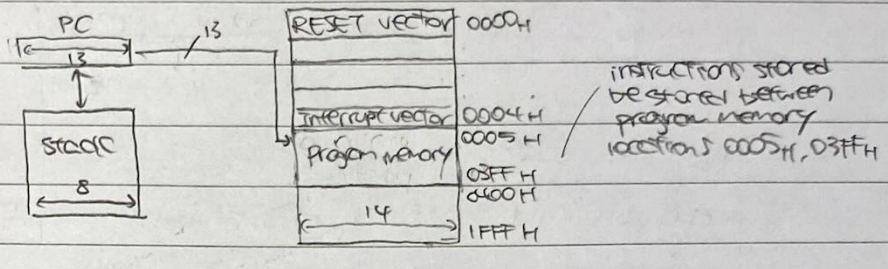
① Special purpose registers.
- ↳ usually 8 bits (the PC is 13 bits, made up of PCL and PCLATH registers)
- ↳ PC holds the address of the next instruction to be executed
- ↳ W holds the data the PIC is working on at the current time (accumulator)
- ↳ STATUS register stores the results of the previous calculation (carry, digit carry, zero flag), as well as power down, time out and register bank information
- ↳ GPIO register holds data either input/output to pins on the chip. [bidirectional]
- ↳ TRISIO register holds the information which bits in the GPIO register are i/ps or o/ps.

② General purpose registers (memory in the Von Neumann architecture)
- ↳ 64 x 8 bit data registers, accessed via File select register (FSR)

Program counter (PC)

- Program counter (PC) is a (13 bit for PIC) register inside the microprocessor that stores the program memory address of the next instruction to be executed.
- The PC is key to the fundamental fetch-decode-execute cycle:
  - ↳ The processor first fetches the instruction from the address stored in the PC.
  - ↳ The fetched instruction is then decoded so that it can be interpreted by the microprocessor.
  - ↳ Once decoded, the instruction is executed and the PC is incremented so that it contains the address of the next instruction.

PIC12F675:



Working register, W

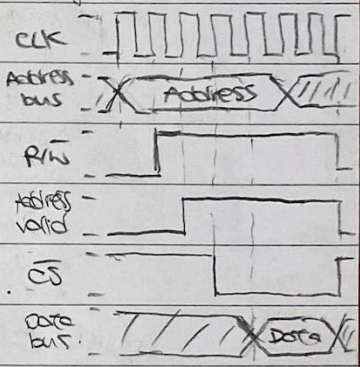
- W holds the data the processor is currently working on
- The data stored can only be an 8-bit word.
- Many PIC instructions take a memory location F and a destination d as argument, the destination can be the location specified by F (d=1) or the working register W (d=0)

Memory and memory chip addressing

Memory read and write operations

- To read (write) data from(to) memory, we
  - ↳ Set address of the memory location on the address bus.
  - ↳ Set the read/write, R/$\bar{W}$ of the control bus HIGH(LOW).
  - ↳ Set the address valid control wire HIGH.

Timing diagram (Read)



- The above together activate the memory chip select $\bar{CS}$ by setting it LOW.
- If R/$\bar{W}$ is HIGH (read operation), the memory chip writes the data bus w/ the data from the location indicated by the address bus.
- Once the data is read (usually into a register on the microprocessor) the control wires are "reset".
- If R/$\bar{W}$ is LOW (write operation), the memory chip writes the data from the data bus into the location indicated by the address bus.
- Once the data is written to the memory, the control wires are "reset".

Memory address capacity

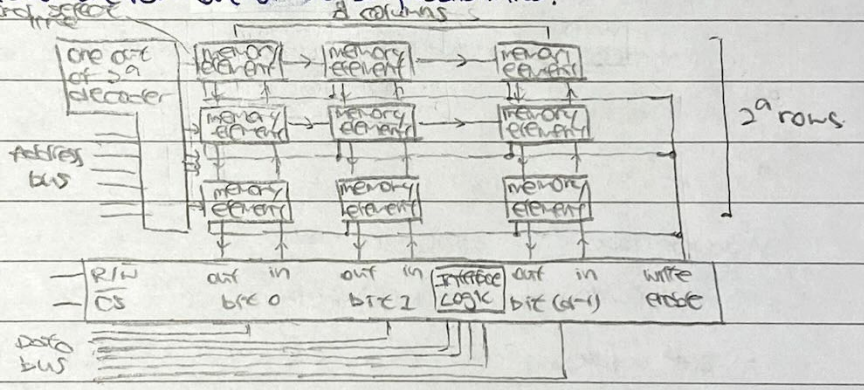
- Let a be the no. of address wires (width of address bus) and d be the no of data wires (width of data wires).
- The total addressed capacity is given by

$$d \times 2^a$$

- 1 kilobinarybyte (kilobyte) is $2^{10} = 1024$ bytes ; 1 megabinary byte (megabyte) is $2^{20} = 1024$ kilobytes $= 1048576$ bytes.
- For the 6800 microprocessor, $a=16$, $d=8$ so the total addressed capacity is 8×2^{16}, or $8 \times 2^{10} \times 2^6 = 64$ kilobytes.

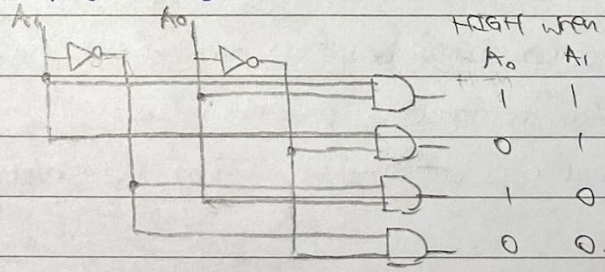
RAM chip architecture.

- Each bit inside the RAM chip has its own memory element, which can be set LOW(0) or HIGH(1)
  - ↳ These memory elements are arranged as a matrix of rows and columns.
  - ↳ There is one row of memory elements for each address and one column for each data wire.
- Address bus signals are passed to a decoder
  - ↳ The decoder determines which row of memory elements corresponds to the address on the bus.
  - ↳ The decoder selects the row out of the 2^a possibilities.



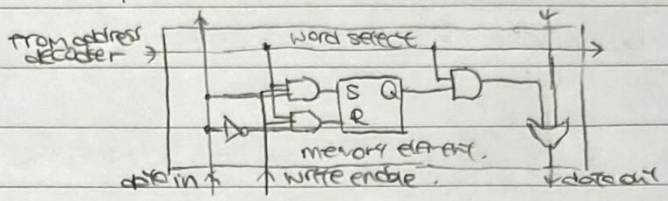
Address decoder (single memory chip)

- The word select lines are driven from the one out of 2^a decoder. → only 1 word select line is driven HIGH at any time.
- The one out of 2^a address decoder selects 1 address line for each combination of address bits.
- A one out of 2^2 decoder is as shown:



Individual memory element.

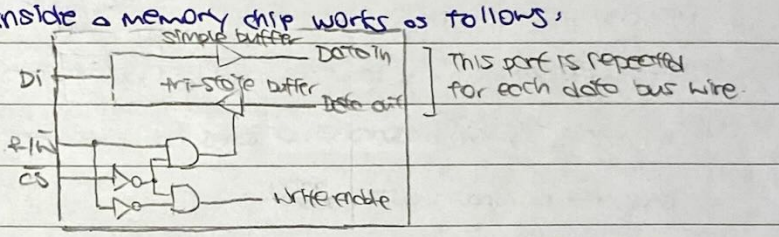
- Each individual memory element is as follows:



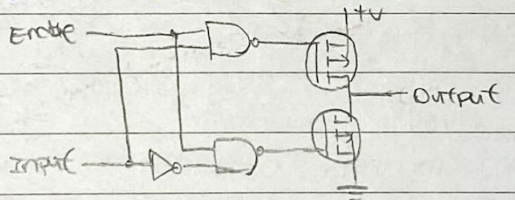
- A particular memory element is selected when the word select line is HIGH.
- When write enable is HIGH, data is written onto the SR bistable; when write enable is LOW, data is read out from the SR bistable.

Data interface at the edge of the chip.

- The interface logic inside a memory chip works as follows:



- We need to select either input or output as the data bus is bidirectional
- For the output, we need to drive the wire + strengthen the signal → tri-state buffer; For the input, we just req. a simple buffer to strengthen the signal.
- Tri-state buffers are connected as follows.



E	i/p	o/p
0	0	floating
0	1	floating
1	0	0
1	1	1

- When the enable is LOW, neither transistor is on and the output is not connected to any voltage by this part of this circuit (high impedance – i/p and o/p disconnected).
- Tri-state buffers are useful in avoiding the loss in data when several logic outputs are connected together (If we use AND/OR gates, the o/ps can affect the i/ps).

Multiple memory chips

- If a single memory chip has a_i address wires and the microprocessor has a_j address wires, we could connect 2^{a_j} memory chips to the microprocessor.
- The a_i LSB address wires $(a_0, a_1 \ldots a_{i-1})$ of the microprocessor are used to address the location within the memory chip.
- The a_j MSB address wires $(a_i, a_{i+1} \ldots a_{j-1})$ of the microprocessor are used to address the memory chip we are selecting.
- e.g.: 1st chip has addresses: $000\,0000_H$ to $000\,FFFF_H$
  2nd chip has addresses: $001\,0000_H$ to $001\,FFFF_H$
  ] Here, $a_i = 4 \times 4 = 16$, $a_j = 4 \times 3 = 12$

Address decoder (Multiple memory chips)

- Other than the one out of 2^{a_i} decoder in each memory chip, we need another one out of 2^j decoder to activate a particular memory chip.

- e.g.: If $a_i = 13$, $a_j = 3$, design the address decoding logic for the 3rd chip (i.e. memory addresses $4000_H - 5FFF_H$).

  $\overline{CS}$ should be set to LOW when the MSBs are 010, i.e. $A_{15}=0$, $A_{14}=1$, $A_{13}=0$

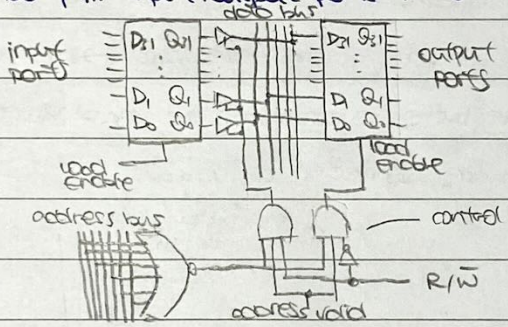
  so the req. logic is $\overline{A_{15}} \cdot A_{14} \cdot \overline{A_{13}}$ ← this is usually AND-ed w/ the address valid wire, i.e. $\overline{A_{15}} \cdot A_{14} \cdot \overline{A_{13}} \cdot \overline{AV}$

↳ $\overline{CS}$ is an active LOW signal.

Memory mapped input/output

- For memory mapped input/output, the input/output ports are treated as memory locations → the microprocessor does not know the difference between a memory chip and an input/output port

- consider mapping both the input/output ports to 00000000_H.



Reading from $0000\,0000_H$ reads from the input port; writing to $0000\,0000_H$ writes to the output port.

↳ If we read a no. at location $0000\,0000_H$, then write at the same location, we get the same no. However, the reverse is not necessarily true.

Other types of memory.

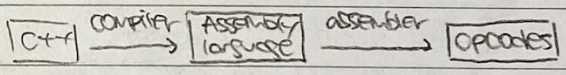
- Random access memory (RAM)
  - ↳ It is readable and writeable, but "forgets" when the power is off (volatile memory).
  - ↳ Static RAM (SRAM) has a simple interface, good storage density, speedy access and low power consumption (when not in active use) → used for fast cache memories on laptops/phones.
  - ↳ Dynamic RAM (DRAM) has a complex interface (content refreshed continuously), consuming power even when not in use, slower than SRAM but has high storage density → used as PC main memory.
- Read only memory (ROM)
  - ↳ ROM contents are set at manufacturing time → can never be changed by the microprocessor.
  - ↳ It is non-volatile and is used to store the necessary instructions for communication between various hardware components

- Erasable programmable read only memory (EPROM)
  - ↳ Cannot be charged by the microprocessor but can be programmed using a device that uses higher voltages than in a normal microprocessor circuit.
  - ↳ Can be erased either using UV light or a higher voltage. EEPROM
- Flash Memory
  - ↳ Non-volatile storage that can be electrically erased and reprogrammed.
  - ↳ Developed from EEPROM – but can be written to and read in blocks (bytes) rather than the entire device.
- More long term storage is achieved through the use of magnetic disks (hard drive) or solid state hard drives.

Microprocessor programming.

programming languages.

- High level languages (C++, Python) are great for application programming but must be translated for the microprocessor to act on the programs.
- Low level languages (machine code) can be hexidecimal opcode sequences or a long sequence of binary 0s and 1s, which can be directly understood by the appropriate microprocessors, but is difficult to write even short programs.
- To make programming easier, an assembler translates an assembly language program into opcodes that are understood by the microprocessor.



Assembly language.

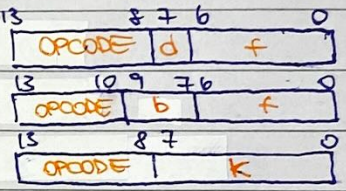
- Assembly language programs are more efficient and req. less memory than programs in high level languages
- There is one assembly language instruction for each opcode in the machine code, giving programmers direct control of the microprocessor.
- It is not complicated for the assembler to translate assembly language into opcodes → this process is quick on computers.
- The PIC only has 35 commands, but it is still possible to build up complex programs from these.

Representing numbers.

- 0x before a no. indicates hexadecimal. (eg. 0x32)
- 0 w/ no x indicates octal from the leading zero. (eg. 024)
- b before the no. indicates binary (eg. b101010)
- otherwise the no. is in decimal (eg. 63).

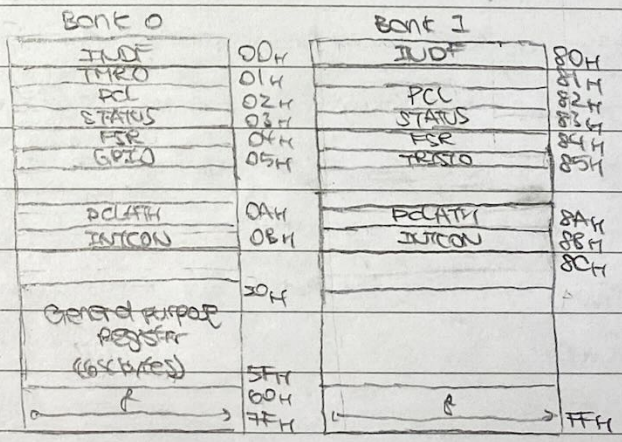
Types of operations

- Assembly language instructions are written using mnemonic code words.
- All PIC instructions are 14 bit words consisting of an opcode and data source information.
- There are 2 main types of instructions
  - ↳ Data processing operations — move, register, arithmetic, logic
  - ↳ Program sequence control operations — unconditional/conditional jump, call, control.
- The operations can be on one of three types of data
  - ↳ Byte oriented: operation is on a byte (8 bits)
  - ↳ Bit oriented: operation is on a single bit of the byte
  - ↳ Literal: operation is on a literal value.



PIC12F675 memory bank

- The PIC12F675 has a memory map organised into 2 banks (Bank 0 and Bank 1) of registers.
- We can select which bank we want to use by clearing (Bank 0) or setting (Bank 1) bit 5 in the STATUS register
- The lower set of address locations in each bank is reserved for special registers.
- The general purpose registers between memory locations 0x20 and 0x5F are used as on chip data memory.

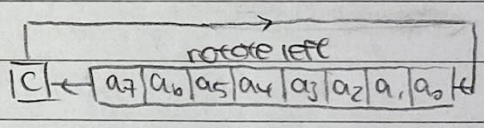


Input/output for PIC12F675

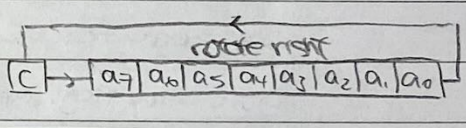
- For the PIC12F675, the GPIO register acts as a 6-bit wide bidirectional port (at memory location 0x05, Bank 0)
- Bit 3 can only be an input; all other pins can either act as an input or an output.
- The corresponding data direction is managed by the TRISIO register (at memory location 0x85, Bank 1)
- Setting a bit in the TRISIO register makes the corresponding bit in GPIO an input; clearing a bit in the TRISIO register makes the corresponding bit in GPIO an output. (TRISIO bit 3 is always set)
- We can set some bits to 1 and some to 0 to have both input and output ports.
* Before inputting data into W from GPIO, we should clear the 2 MSB in W as we only have 6 inputs. (at most).

Special uses of commands:

- andwf can be used to round numbers
  - ↳ Round to the nearest even no. by AND-ing w/ b'11111110'
  - ↳ Round to the nearest odd no. by AND-ing w/ b'11111110' then increment by 1.
- rlf can be used to determine whether the original no. was +ve or -ve in 2's complement or multiply by 2 (sometimes)
  - ↳ check C flag. 0 → +ve; 1 → -ve.
  - ↳ NO. is multiplied by 2 (sometimes — check C and MSB)



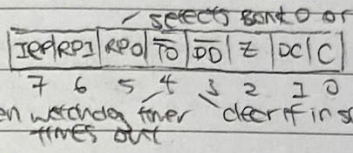
- rrf can be used to determine whether the original no. was odd or even or divide by 2 (sometimes)
  - ↳ check C flag. 0 → even; 1 → odd
  - ↳ NO. is divided by 2 (sometimes — check C and LSB)



STATUS register, flags, tests and flow charts

STATUS register

- Several bits in the STATUS register hold information about the result of the last arithmetic or logic operation → can be used in a conditional instruction.
- Other bits contain power down, time out and register bank information.
- The STATUS register is at memory location 0x03, Bank 0 and 0x83 Bank 1.



Tests and conditional instructions.

- Several instructions can test a bit in a register or data word.

↳ btfsc F,b ↳ btfss F,b	often used to test the C, DC and Z flags in the STATUS register
↳ decfsz F,d ↳ incfsz F,d.	often used to generate a delay or in timers

Carry flag, C

- The C flag is set when there is a carry from the MSB (bit 7) during a calculation.
- If we have an addition

$$F \text{ plus } W \rightarrow W_{after}$$

then the carry flag will be determined by the Boolean expression

$$C = F_7 \cdot W_7 + (F_7 + W_7) \cdot \overline{W_{7\,after}}$$

both MSB are 1 — either MSB is 1 — MSB of the result is 0

Digit carry flag, DC

- A byte is made up of 2 nibbles (4 bits) — there is a lower nibble (bits 0–3) and a upper nibble (bits 4–7)
- The DC flag is set when there is carry from the lower to the higher nibble during a calculation.
- If we have an addition

$$F \text{ plus } W \rightarrow W_{after}$$

then the digit carry flag will be determined by the Boolean expression

$$DC = F_3 \cdot W_3 + (F_3 + W_3) \cdot \overline{W_{3\,after}}$$

both bit 3 are 1 — either bit 3 is 1 — bit 3 of the result is 0

Zero flag, Z

- The Z flag is set when the result of an instruction is zero
- If we have an addition:

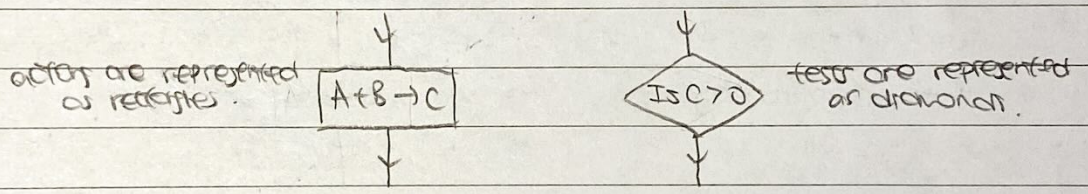
$$F \text{ plus } W \rightarrow \text{Wafter}$$

then the zero flag will be set if

$$\boxed{\text{Wafter} = 0}$$

Flowcharts for assembly programs

- Flowcharts are a graphical technique for designing short programs
  - ↳ ✓ : encourage design of the overall algorithm
  - ↳ ✓ : independent of microprocessor architecture / instruction set
  - ↳ ✓ : easier to read than machine code / assembly language.
  - ↳ x : modern programming techniques in high level languages are quicker to use
  - ↳ x : the precise way the processor instructions are used are important for machine code programming → flowcharts not useful.
- the conventions for flow charts are as follows:

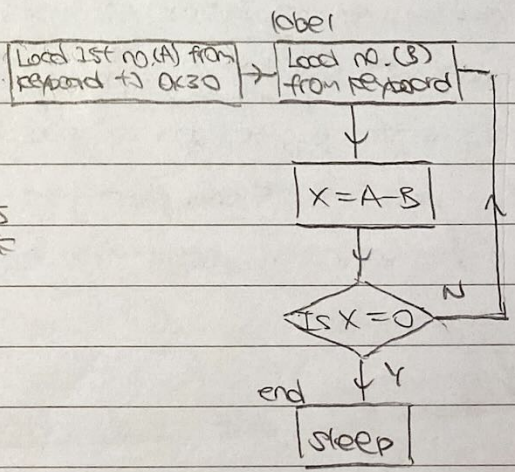


goto, loops and the use of labels

- In conditional branches, we can use goto to jump to a label.
- The goto instruction is quite useful for making loops.

-eg:

```
label   movf GPIO, 0x30;
        movf GPIO, W ;
        subwf 0x30, W ;
        btfss STATUS, Z ;
        goto label ;
        goto end ;
end     sleep;
```



loops back

* Here, we assume TRISIO is set so all GPIO pins are inputs.

Program run time

- For most PIC processors, each instruction takes 1 internal clock cycle to execute.
- There are 2 main exceptions that take 2 internal clock cycles to execute
  - ↳ Instructions which need to replace PC contents (goto, call, return etc.)
  - ↳ Test/conditional branch instructions — if the conditional branch does skip, it discards the next instruction and executes a nop (o/w only 1 IN)

Generating a delay

- It is often useful to be able to generate a delay whilst waiting for an event to happen, (e.g. turn on LED ON for a fixed time)
- Consider the delay program that uses the decfsz instruction.

```
        movlw 0xFF;     1N
        movwf 0x30;     1N
loop    decfsz 0x30;    1N for first 254, 2N for final
        goto loop;      2N
        sleep.          1N
```

Total runtime = $1+1+1\times254+2+2\times254+1 = 767$ clock cycles.

If the microprocessor has a 4MHz clock, total execution time $\frac{767}{4\times10^6} = 191.5\,\mu s$.

File register indirect addressing

- The File Select Register (FSR) is used for indirect addressing (at memory location 0x04, Bank 0 and 0x84, Bank 1)
- If a file register address is loaded into FSR, the contents of that register can be read through the Indirect File Register (INDF), (at memory location 0x01, Bank 0 and 0x81, Bank 1).
- This method can be used to access a set of data in memory locations by reading/writing the data via INDF and selecting the next location by incrementing FSR.
- This is particularly useful for storing a sequence of data that has been read in through a port.

-e.g Load a set of file registers 0x20–0x2F w/ dummy data 0xAA

```
    movlw 0x20;
    movwf FSR;
    movlw 0xAA;
loop movwf INDF;
    incf FSR;
    btfss FSR,4;
    goto loop;
    sleep;
```

we start at 0010 0000

and end at 0010 1111

→ look for 0011 0000 → bit test bit 4.

Assembling into machine code

- the data block has a list of commands and its machine code equivalent.

  e.g. movwf F is 00 0000 1fff ffff so movwf 0x30 is 00 0000 1011 0000

- For a program, each 14-bit instruction is stored in a program memory location

↓ Program memory location has a 13-bit address and 14-bit content;

File register memory location has a 8-bit address and 8-bit content

Stack, subroutines, interrupts and the RESET pin

The stack – basics

- The stack is a data structure that allows data to be stored and retrieved in an organised way.
- It is "last in, first out" — last item placed on a stack is the first item to be removed (as opposed to queue, which is "first in, first out")
- When data is added to the stack, it is PUSHed onto the stack; when data is removed from the stack, it is POPped off the stack.
- In many microprocessors, we can use assembly instructions to PUSH and POP items to/from the stack.
- In a PUSH operation, the contents of one or more registers will be placed on the stack. The memory address location where the 1st item is stored is held in the stack pointer (SP) (SP is another register.)
- In a POP operation, we load content into one or more registers and adjust the SP to reflect the freeing up of memory.

The PIC12F675 Stack.

- The PIC12F675 has an 8-level deep, 13 bit wide stack.
- the stack space is not part of either program or data (file) space and the SP is not readable or writeable (i.e. no PUSH or POP commands)
- Instead, the stack is automatically used for sub-routines ← so no need to worry for day-to-day programming.
  - ↳ The PC is PUSHed onto the stack when a call instruction is executed
  - ↳ The stack is POPped in the event of a return, retlw or a retfie instruction execution.

Sub-routines (Functions)

- Sub-routines are used to carry out discrete program functions → allow programs to be written in manageable self-contained blocks which can be executed as req.
- call is used to jump to a sub-routine and return to terminate it and return to the main program
- retlw is the same as return but also stores literal value in W.
- The call command has the address of the sub-routine as an argument (The main program PC contents are pushed onto the stack and replaced w/ the sub-routine address).

Interrupts

- Interrupts are generated by an asynchronous event (i.e. not linked to program timing)
- When an interrupt is called, the Interrupt Service Routine (ISR) is run, PC is pushed onto the stack and is set to 0x0004 (Interrupt vector)
- When the ISR finishes w/ the retfie instruction, the PC is reloaded from the stack and the original program must resume as if nothing has happened.

The $\overline{\text{RESET}}$ pin.

- Microprocessors have a $\overline{\text{RESET}}$ input that when activated initialises the processor and starts it running in a predetermined and safe fashion. (PC is reinitialised to 0x0000)
- A reset button can be connected that brings the $\overline{\text{RESET}}$ pin voltage to LOW and hence restart the microprocessor → emergency reboots
- When the power is switched on, suitable circuitry can ensure that the $\overline{\text{RESET}}$ pin voltage is held at LOW for a short period → start up in a predefined state.

the actual time depends on how long it takes the processor to initiate the reset sequence.