Digital Logic

D.K. Sharma, D. Chakraborty, K. Chatterjee, B.G. Fernandes,
J. John, P.C. Pandey, N.S. Shiradkar, K.R. Tuckley

EE Department
IIT Bombay, Mumbai

Book References

"Digital Fundamentals" by Thomas L. Floyd, Pearson Education.
"Digital Design" by M. Morris Mano, Pearson Education.

January 17, 2024

What is Digital Logic?

- The term “digital” is derived from digit, a symbol (like 0 to 9) in a number system – (and ultimately, from fingers which are used for counting).
- Thus, digital representation is a natural fit for things which are discrete and countable.
  For example – natural numbers are discrete, so natural numbers can be represented by digital symbols.
- We are used to representation of numbers using ten symbols – this is the familiar decimal number system. However, circuit implementation of a digital system with just two symbols – a *binary* digital system – is much more robust and reliable.
- In a circuit, we can associate one of the symbols, say ‘1’ with the highest voltage (the supply voltage) and the other, say ‘0’ with the lowest voltage (ground). Now the two digits can be reliably represented and will not be confused with each other even in the presence of very high noise.

What is Digital Logic?

- The term **binary digit** (a ‘1’ or a ‘0’) is contracted to a **bit**.
- The binary digital system is also a natural fit for logic, in which statements can be either ‘True’ or ‘False’.
  Thus we can use a bit to represent the truth or otherwise of a logical statement.
- This has led to the widely used terminology – **Digital Logic**.
- Circuits and systems which implement Digital Logic are used for computer design, digital communications, industrial controls and many other applications.
- Binary representation is not universal – there are systems which divide the span between the supply voltage and ground in multiple (say 4) regions. However, binary representation is used in an overwhelming majority of digital systems.

Representing Natural Numbers

- To represent bigger numbers, we use multiple bits with implied place values of powers of 2, just as we do for decimal numbers. If we use enough bits, we can represent a wide range of numbers.
- For example, we can represent the decimal number 10 as 1010 ($= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 0 + 2 + 0 = 10$).
- The number whose powers are taken as place values is called the **base** for a number representation.
- Thus the familiar decimal numbers are base 10 numbers and binary representation is base 2.
- We shall discuss the procedure for converting numbers from decimal to binary in a subsequent lecture.

Representing Natural Numbers

- The number of bits (base 2) required to represent a number is much larger than the decimal digits required in a base 10 representation.
- So the representation of somewhat larger numbers can become awkward. For example, decimal 1000 will be represented as 1111101000.
- It is conventional to group the bits 4 by 4 from the least significant side using single symbols.
- These symbols should represent numbers from 0 to 15. We use the conventional decimal symbols for 0 to 9 and use A, B, C, D, E and F for 10, 11, 12, 13, 14 and 15 respectively. This system is called **Hexadecimal** or base 16.
- Thus 1000 (decimal) will be represented as (0011 | 1110 | 1000) =3E8 in hexadecimal. ($3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0 = 768 + 224 + 8 = 1000$)

What about things which are not discrete?

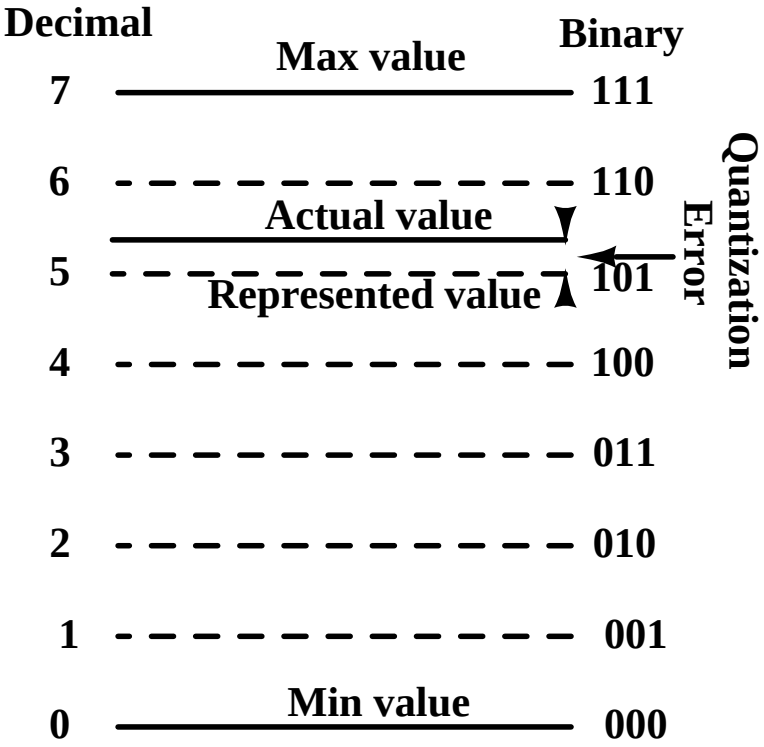
If we use enough bits, we can represent a wide range of numbers using binary digits or bits.

But how can we represent continuous things – like voltage, current or light intensity?

- Continuous quantities cannot be represented exactly by digital values. However, if we are willing to use an adequate number of bits, we can make a close approximation to continuous values using digital representation.
- We divide the full range of values into a number of intervals separated by discrete levels. Now we can approximate the continuous value by the number associated with the discrete level closest to the actual value.
- The inaccuracy of representation of a continuous value by a digital representation is called “quantization error”.

Digital representation of Continuous values

We approximate the actual value by the discrete level closest to it.



- The difference between the actual value and the represented value is the quantization error.
- In the worst case, this error can be equal to half the interval width.
- If we divide the range of values into a larger number of intervals, each interval will be narrower and the quantization error will be smaller.

However, the number of bits required for labeling the intervals will be more if we divide the range into a larger number of intervals.

Notice that close but unequal values may be represented by the same digital number.

Truth tables

- Binary digits ‘1’ and ‘0’ may be used to represent ‘True’ and ‘False’ values.
- There can be a logical relationship between different logical statements. For example:
  - The rear right light of a car should blink if we want to turn right.
  - Both rear lights should blink if we are in emergency mode.

So when should the blink control of the rear right light be turned on?

Clearly, when either or both of these events (intention to turn right or being in an emergency situation) is ‘True’.

We can express this logical relationship using an exhaustive enumeration of all possible combinations of causing events, since the number of combinations of such events is limited.

Tables containing such enumeration are called “Truth tables”.

Truth tables and Logic Functions

Consider the truth table for the blinking of the rear right light.

Right Turn?	Emergency?	Blink rear right light?
False	False	False
False	True	True
True	False	True
True	True	True

This table exhausts all possible logic value combinations for the causing events “Right Turn?” and “Emergency?”. The table represents a unique logic function, which in this case is termed as “OR”.

If we represent ‘True’ by 1 and ‘False’ by 0, the truth table for the OR function of two logical variables A and B can be written as shown on the right.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Truth tables and Logic Functions

How many distinct logic functions are possible for n input variables?

- The truth table must enumerate all possible combinations of input variables. So it will have 2^n rows.
- The function value can be either 0 or 1 in each row. Thus there are 2 independent choices per row.
- So the number of distinct truth tables is $= 2^{\text{no. of rows}} = 2^{2^n}$.
- Each distinct truth table represents a different logic function. (Though some of these functions could be trivial).
- Rules of manipulation of binary valued objects and logic functions which operate on them constitute an algebra. This algebra is known as “Boolean Algebra”, named after the English mathematician George Boole.

George Boole, “Mathematical Analysis of Logic”, 1847, and “An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities”, 1854

Truth tables and Logic Functions

Consider functions of a single variable – there can be $2^{2^1} = 4$ tables. Labeling input as a Boolean variable A, these are:

A	Out
0	0
1	0

A	Out
0	1
1	0

A	Out
0	0
1	1

A	Out
0	1
1	1

- The output in the first table is always 0. This is thus a trivial function which is always 0 independent of the input.
- In the second truth table, the output is always opposite to the input. This function is called an invert function or a **NOT** logical function. Given a binary variable A, NOT A is represented as $\overline{A}$.
- In the third truth table, the output is the same as the input. This is a buffer function, where Out $= A$.
- In the last truth table, the output is always 1. This is also a trivial function which is always 1, independent of the input.

Logic Functions of two variables

For 2 input variables, we can have $2^{2^2} = 16$ distinct logic functions.
A few important ones are described below:

A	B	OR	NOR	AND	NAND	XOR	XNOR
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1

The output of OR is '1' if either or both inputs are '1'.

The output for the AND function is '1' only if both inputs are '1'.

The output for the XOR (exclusive OR) function is '1' only if exactly one of the inputs is '1'. It is like the OR function but excludes the case when both inputs are '1' – hence the name.

NOR, NAND and XNOR are inverted values of OR, AND and XOR respectively.

Boolean Algebra: Terminology

George Boole provided a mathematical foundation to logic through two monographs published in 1847 and 1854.

- Boolean algebra uses the symbol "+" for the OR logical function and the symbol "$\cdot$" for AND.
- We use a bar on top of a variable to denote the NOT function. Thus, NOT $x = \overline{x}$. Since a bar is not easily typeset, a $'$ sign is also used to indicate the NOT operation. Thus x' is an alternative to $\overline{x}$.
- NOT, OR and AND are the basic functions of Boolean algebra. The XOR function is represented as $\oplus$.
- If x is True, $\overline{\text{x}}$ must be False and if x is False, $\overline{\text{x}}$ must be True. Therefore one of x and $\overline{x}$ must always be True and one of x and $\overline{x}$ must always be False.

George Boole, "Mathematical Analysis of Logic", 1847, and "An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities", 1854

Boolean Algebra: logical relationships

We can verify some identities for Boolean functions quite easily.
For example: $x + 0 = x, \quad x + 1 = 1, \quad x \cdot 0 = 0, \quad x \cdot 1 = x$
(Since OR is True if either input is True while AND is False if either input is False).

Some other identities for Boolean functions can be verified easily from the truth tables.

A	B	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

OR as well as AND outputs are 0 and 1 respectively for both inputs being 0 or 1.
Thus $x + x = x, \quad x.x = x$

$x + \overline{x} = 1, \quad x.\overline{x} = 0$ since one of x and $\overline{x}$ must be True and the other must be False.

Boolean Function Identities

Obviously, inverting a Boolean variable twice will give back the original value. So $\overline{\overline{x}} = x$.

A	B	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

From the truth tables, we can see that the OR and AND outputs are symmetric with respect to the two inputs.
so $x + y = y + x, \quad x.y = y.x$
(OR and AND functions are commutative).

Using these results, further identities can be proven algebraically.

What is the least number of logical relations which we can assume as obvious and from which all the other properties of Boolean functions can be derived algebraically?

This minimal set of results is called “axioms” or “postulates”.

Huntington Postulates

Huntington formulated postulates for Boolean algebra in 1904. Using these, many of the useful theorems of Boolean logic can be derived (without having to take further recourse to truth tables). These are:

Identity:

$A + 0 = A, \quad A \cdot 1 = A$

complement:

$A + \overline{A} = 1, \quad A \cdot \overline{A} = 0$

commutation:

$A + B = B + A, \quad A \cdot B = B \cdot A$

distributivity:

$A + (B \cdot C) = (A + B) \cdot (A + C),$

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

Most of these properties are shared with ordinary algebra.

Only the distribution over the "$\cdot$" operator – $(A + (B \cdot C) = (A + B) \cdot (A + C))$ is peculiar to Boolean algebra.

Notice the duality between $+$ and $\cdot$ operators. The dual postulate is obtained by changing $+$ to $\cdot$ and changing the identity from 0/1 to 1/0.

Some theorems of Boolean Algebra

Once the postulates have been verified (say using truth tables), we can proceed to prove several useful theorems using these postulates:

$x + x = x, \quad x \cdot x = x$
$x + 1 = 1, \quad x \cdot 0 = 0$

Associativity

$x + (y + z) = (x + y) + z,$
$x \cdot (y \cdot z) = (x \cdot y) \cdot z$

Absorption:

$x + x \cdot y = x$

x "absorbs" any **added** term of the form x · (anything).

$x \cdot (x + y) = x$

Here x "absorbs" any **multiplied** term of the form (x + anything).

Notice that associativity is not a postulate, but can be proved as a theorem from the postulates described earlier.

DeMorgan's theorems

DeMorgan's theorems deal with the effect of the NOT operation on OR and AND logic functions. These state that:

$$\overline{x + y} = \overline{x} \,.\, \overline{y} \quad \text{and} \quad \overline{x.y} = \overline{x} + \overline{y}$$

Essentially, when applying the NOT operator to an expression, all variables get converted to their 'bar' versions (known as complements), all '+' operators get converted to '.' operators and all '.' operators become '+' operators.

Order of precedence:

In a an expression involving logic variables and functions, the order of precedence for evaluation is:

i) parentheses, ii) NOT, iii) AND, iv) OR operation.

From Truth Tables to Logic Expressions

Every truth table is equivalent to a logic expression.

How do we associate a given truth table with a logic expression?

- A truth table must enumerate all possible combinations of logic values for the input variables.
- Select all rows which have a '1' as the function value.
  If the input combination falls in any of these rows, the function evaluates to '1'.
- Therefore the output must be the OR of all these rows.

But how do we associate a row of the truth table with a logic expression?

From Truth Tables to Logic Expressions

Each row of the truth table may be represented by an AND function of each of the input variables or their complements.

- Consider a truth table for 3 input variables A, B and C. A particular row may contain the values 1 0 1 for these three inputs. These values uniquely identify this row of the truth table.
- We know that the AND of three Boolean variables can be '1' only if **all** of them are '1'. (If any one is '0', the product would be '0').
- So the row is characterised by the AND of all variables whose value is '1' and the complement of all variables whose value is '0' in this row.
- Thus this row would be represented as $A \cdot \overline{B} \cdot C$.

Identification of rows by the product of all input variables or their complements (as used for uniquely identifying a row of the truth table) is called a **minterm**.

From Truth Tables to Logic Expressions

The logic function is an OR of all those minterms for which the function evaluates to '1'.

Let us illustrate it with an example with 3 input variables. We show two independent logic functions F1 and F2 of these 3 variables.

A	B	C	F1	F2
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The first function has the value '1' for rows with values of A, B, C as $001, 010, 011$ and 111.

So the function is:
$\overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$.

The second function is '1' for ABC values of $000, 010, 011, 101, 110$ and 111.

Correspondingly it can be written as
$\overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$

From Truth Tables to Logic Expressions

This method of associating a truth table with a logical expression is general and can be used with any number of input variables.

- A logical expression which identifies a row of the truth table as a product is called a "**minterm**". Each "minterm" is the product of *all* input variables – either in their original form or in the complement form.
- The final logic function representing the truth table is then the OR (or sum) of all "minterms" for which the function has the value '1'.
- The method of expressing the logic function using minterms is a standard (or **canonical**) form which is also known as a **sum of products** form.
- A minterm expression for the *i*'th row in a truth table is represented as $\mathbf{m}_i$.

Canonical: which follows canons or well defined rules.

From Truth Tables to Logic Expressions

Recall that we had said that we can represent numbers using binary digits (bits) by allocating place values to these bits which are powers of 2, just like decimal numbers.
Now consider the functions F1 and F2 of 3 variables which we had used as examples earlier.

A	B	C	F1	F2
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- The rows for which the output of the first function is '1' are: $001, 010, 011$ and 111.
- If we interpret the row as a decimal number derived with place values which are powers of 2, we can identify these rows as no. $1, 2, 3$ and 7.
- Using this convention, we can also write the function as $m_1 + m_2 + m_3 + m_7$, where m_i represents the i'th minterm.

From Truth Tables to Logic Expressions

We had remarked on the duality of theorems in Boolean logic. That provides us with another way of associating a logic expression with a truth table.

Select all rows which have a '0' as the result. If the input combination falls in *any* of these rows, the output is '0'. The output must be the AND of all these rows.

How do we uniquely represent a row for which the function evaluates to '0'?

- Consider again a truth table for 3 input variables A, B, and C, with a row which contains the values 1 0 1 for these three inputs.
- We know that the OR of three Boolean variables can be '0' only if **all** of them are '0'. (If any one is '1', the 'sum' would be '1').
- So the row is characterised by the OR of all input variables whose value is '0' and the complement of all input variables whose value is '1' in this row.

From Truth Tables to Logic Expressions

This alternative way of finding a logical expression to represent a row of the truth table for which the function evaluates to '0' is called a **Maxterm** and represented as $\mathbf{M}_i$.

Let us use the same example as the one we had taken earlier (with a truth table for two functions F1 and F2 of three input variables).

A	B	C	F1	F2
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The rows for which the value of the first function (F1) is '0' are: $000, 100, 101$ and 110.

These are represented by Maxterms: $A+B+C$, $\overline{A}+B+C$, $\overline{A}+B+\overline{C}$, and $\overline{A}+\overline{B}+C$, respectively.

So, F1 can be represented in the alternative form as:

$$(A+B+C)\cdot(\overline{A}+B+C)(\overline{A}+B+\overline{C})\cdot(\overline{A}+\overline{B}+C)$$

From Truth Tables to Logic Expressions

For the function F2, we have:

A	B	C	F1	F2
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The rows for which the value of the second function (F2) is '0' are: 001 and 100.

These are represented by Maxterms: $A + B + \overline{C}$ and $\overline{A} + B + C$.

So, F2 can be represented in the alternative form as:

$$(A + B + \overline{C}) \cdot (\overline{A} + B + C)$$

This is also a **canonical** form and is known as the **product of sums** form.

Logic Reduction

- Any logical combination can be expressed using truth tables. We have seen that a truth table can be reduced to **canonical** forms such as "**sum of products**" or "**product of sums**". These provide standard ways of representing any logic function.
- However, these logic expressions are not in their simplest form.
- When logic is implemented in hardware, each operation corresponds to some logic element which contributes to the cost as well as the delay in evaluating the expression.
- Thus there is a strong motivation to simplify these expressions as much as we can.
- Using Boolean algebra identities, it is often possible to reduce these expression to a much simpler form, which is then easier to implement.

Logic Reduction

As an example of logic simplification, let us take the function F1 which we had derived in the sum of products form earlier.
$F1 = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$

Using the logic identity $x + x = x$, we can write the above function as:
$F1 = (\overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C) + (\overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C) + (\overline{A} \cdot B \cdot C + A \cdot B \cdot C)$
where we have replicated the third term and added it to first and second terms. This can be simplified to
$\overline{A} \cdot (\overline{B} + B) \cdot C + \overline{A} \cdot B \cdot (\overline{C} + C) + (\overline{A} + A) \cdot B \cdot C$

Using the identity $x + \overline{x} = 1$, we can write the above as
$\overline{A} \cdot C \cdot 1 + \overline{A} \cdot B \cdot 1 + 1 \cdot B \cdot C = \overline{A} \cdot C + \overline{A} \cdot B + B \cdot C = \overline{A} \cdot (B + C) + B \cdot C$

It is easy to verify that this much simpler function evaluates to '1' for ABC combinations of (001), (010), (011) and (111), which agree with the truth table from which we had derived the sum of products function.

Logic Reduction

Simplification of canonical forms like sum of products or product of sums is not always obvious and it is not clear what is the best strategy for simplifying the expression.

A graphical solution which follows a standard procedure and works well for a small number of input variables was suggested by Maurice Karnaugh.

This method is known as the **Karnaugh Map** method. (The name is often shortened to K-map).

Karnaugh, Maurice (November 1953) [1953-04-23, 1953-03-17].
“The Map Method for Synthesis of Combinational Logic Circuits” (PDF).
Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics, 72 (5): 593-599.

doi:10.1109/TCE.1953.6371932.

Logic Reduction by Karnaugh Maps

- In this method, we translate the truth table to a 2D map, where each row and column corresponds to combinations of a small number of input variables.
- The row and columns of a Karnaugh Map are arranged such that only one variable changes from a row/column to the adjacent one. (This is known as **Gray code**).
- For example, a Karnaugh map for functions of 4 variables is shown below:

CD↓ AB→	00	01	11	10
00				
01				
11				
10				

The function values (‘1’ or ‘0’) corresponding to respective input combination are now transferred to the cells at the crossing of AB and CD values.

Logic Reduction by Karnaugh Maps

Minterms represented by each cell

$CD\downarrow$ $AB\rightarrow$	00	01	11	10
00	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$	$\overline{A}\,B\,\overline{C}\,\overline{D}$	$A\,B\,\overline{C}\,\overline{D}$	$A\,\overline{B}\,\overline{C}\,\overline{D}$
01	$\overline{A}\,\overline{B}\,\overline{C}\,D$	$\overline{A}\,B\,\overline{C}\,D$	$A\,B\,\overline{C}\,D$	$A\,\overline{B}\,\overline{C}\,D$
11	$\overline{A}\,\overline{B}\,C\,D$	$\overline{A}\,B\,C\,D$	$A\,B\,C\,D$	$A\,\overline{B}\,C\,D$
10	$\overline{A}\,\overline{B}\,C\,\overline{D}$	$\overline{A}\,B\,C\,\overline{D}$	$A\,B\,C\,\overline{D}$	$A\,\overline{B}\,C\,\overline{D}$

- Notice that nearest neighbour cells differ in a single term being with or without bar.
- If the function has the value ‘1’ in any two adjoining cells, these two terms will be included in the OR expression.
- While taking the OR of these terms, we can take the identical terms as common and the differing terms will have the form $(x + \overline{x})$.
- Thus the differing terms will get eliminated.

Logic Reduction by Karnaugh Maps

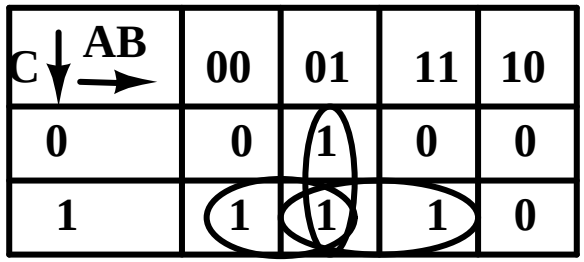
This leads to the following procedure for logic minimization of sum of product forms:

- Arrange cells as shown earlier, such that the minterms represented by the nearest row/column differ in only one input variable being with or without bar.
- Left and right extreme columns and top and bottom rows are considered as being adjacent, since these also differ in a single input variable being with or without bar.
- Circle the largest possible groups of '1' values covering 2^n rows/columns.
- Each circle will eliminate input variables for which both '0' and '1' values are included in the circle.
- Each circle gives a term in the final logic expression. Any isolated '1's will have their full minterm included.

Logic Reduction by Karnaugh Maps

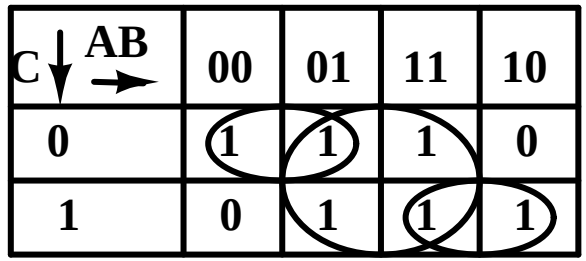
Let us illustrate the procedure by minimizing the two functions we had taken as examples earlier.

A	B	C	F1	F2
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$F1 = \overline{A}\,B + \overline{A}\,C + B\,C$

Vertical ellipse eliminates C
Left horizontal eliminates B
Right horizontal eliminates A



$F2 = \overline{A}\,\overline{C} + B + AC$

Left horizontal eliminates B
Big ellipse eliminates A and C
Right horizontal eliminates B

Notice that F1 expression is the same as the one derived earlier using Boolean algebra.

Canonical expressions in the product of sums form can be similarly minimized by grouping 0's in the Karnaugh map.

Karnaugh Map example: Adders

The truth table for arithmetic addition of two bits is:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Karnaugh Map: Sum

B↓ A→	0	1
0	0	1
1	1	0

Karnaugh Map: Carry

B↓ A→	0	1
0	0	0
1	0	1

No minimisation is possible using Karnaugh Maps.

$\text{sum} = A \cdot \overline{B} + B \cdot \overline{A} = A \oplus B, \qquad \text{carry} = A \cdot B$

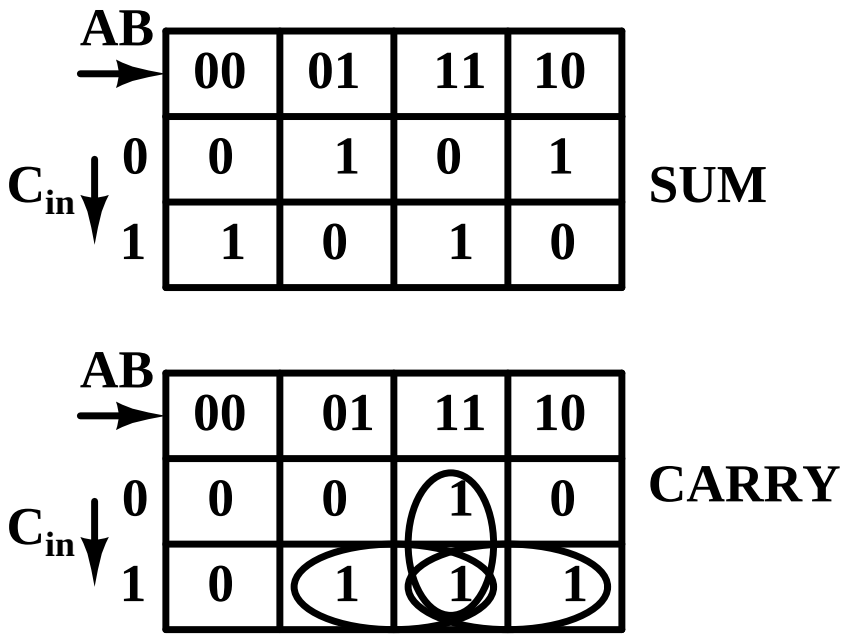
- What do we do with the carry?
- Obviously, it must be added to more significant bits.
- So we need an adder with *three* inputs.

Full Adder

Full Adder Truth Table:

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

It gives the following K maps:



$$\text{sum} = \overline{A} \cdot \overline{B} \cdot C_{in} + \overline{A} \cdot B \cdot \overline{C_{in}} + A \cdot B \cdot C_{in} + A \cdot \overline{B} \cdot \overline{C_{in}} = A \oplus B \oplus C$$

$$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A = A \cdot B + C_{in} \cdot (A + B)$$

- In this lecture, we have learnt about Boolean algebra, its axioms and theorems and their use in expressing and manipulating logic expressions.
- We have learnt about Karnaugh maps – a graphical method for logic minimization.
- In the next lecture, we shall learn:
  - how digital logic is implemented in hardware,
  - number systems using binary and hexadecimal representations, and
  - digital to analog conversion (DAC) and analog to digital conversion (ADC).

Book References

"Digital Fundamentals" by Thomas L. Floyd, Pearson Education.
"Digital Design" by M. Morris Mano, Pearson Education.

Digital Logic: Implementation

D.K. Sharma, D. Chakraborty, K. Chatterjee, B.G. Fernandes,
J. John, P.C. Pandey, N.S. Shiradkar, K.R. Tuckley

EE Department
IIT Bombay, Mumbai

Book References

"Digital Fundamentals" by Thomas L. Floyd, Pearson Education.
"Digital Design" by M. Morris Mano, Pearson Education.

January 17, 2024

Review

In the previous lecture, we had learnt about Boolean algebra, its axioms and theorems and their use in expressing and manipulating logic expressions.
We had also learnt how to minimize logic expressions to simplify their implementation.

In this lecture, we discuss:

- combinational and sequential logic,
- how digital logic is implemented in hardware,
- number systems using binary and hexadecimal representations,
- digital to analog conversion (DAC) and analog to digital conversion (ADC).

Book References

"Digital Fundamentals" by Thomas L. Floyd, Pearson Education.
"Digital Design" by M. Morris Mano, Pearson Education.

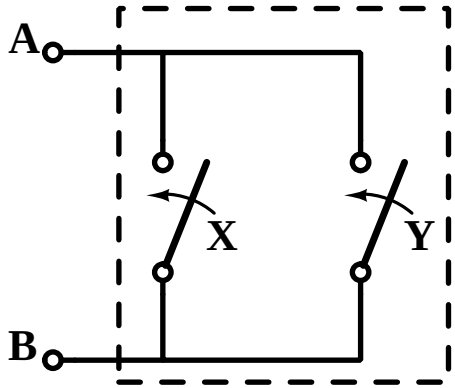
Combinational and Sequential Logic

- The logic functions we have seen up to now produce an output as a function of the combination of current values of input variables. Logic of this kind is known as **Combinational Logic**.
- However, some applications require the output to generate a **sequence** of values as a function of a **sequence** of values at the input. Logic functions of this kind constitute **sequential** logic.
- Sequential logic requires
  - A way to mark points in time which separate the “previous” value from the “current value” on any any input or output, and
  - means of storing information about previous values of data (**memory)**.
- The signal, which marks points in time separating different elements of the input and output sequences, is known as the **clock**.

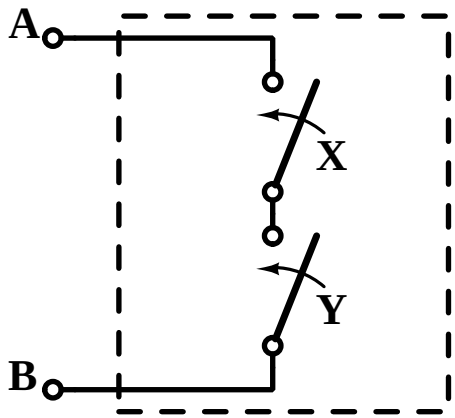
Implementing Digital Logic

How can we implement Digital Logic in hardware?
One possibility is to use electronic devices as switches. Let us see how switches may be used to implement logic functions.

First, consider two switches connected in parallel across nodes A and B.



- Obviously, node A is connected to node B if either or both switches are ON.
- This can be used to implement the logical OR.



- When the switches are connected in series, node A is connected to node B only if *both* switches are ON.
- This can be used for implementing AND logic.

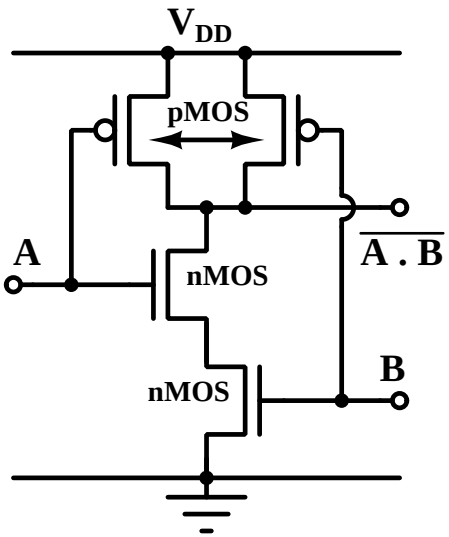
Logic Gates

- In the early days, relays were used as controllable switches to implement logic functions.
- More recently, logic circuits have been implemented using diodes, bipolar transistors and MOS transistors.
- These days, digital circuits are implemented using a combination of n and p channel MOS transistors. This is known as complementary MOS or CMOS logic.
- TTL (transistor-transistor-logic) was the dominant technology for a long time and is still in use. It used bipolar transistors and operates with a 5 V supply. Most TTL gates have type numbers of the form 74xx.
- CMOS B series uses complementary MOS transistors and can operate with supply voltages of 3 to 15V. Typical type numbers are CD 4xxx.
- A TTL compatible logic series implemented with CMOS is also available and carries type numbers like 74Cxx.

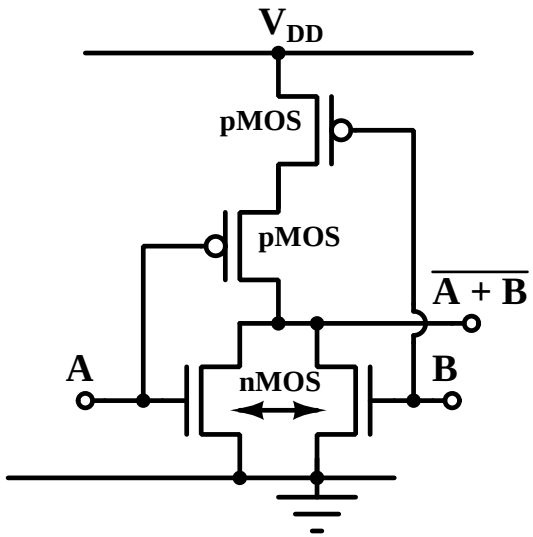
CMOS Logic Gates

CMOS gates implement switch based logic that we had discussed earlier. A '1' at the input turns the nMOS ON and the pMOS OFF, while a '0' at the input turns the nMOS OFF and the pMOS ON.

CMOS NAND



CMOS NOR

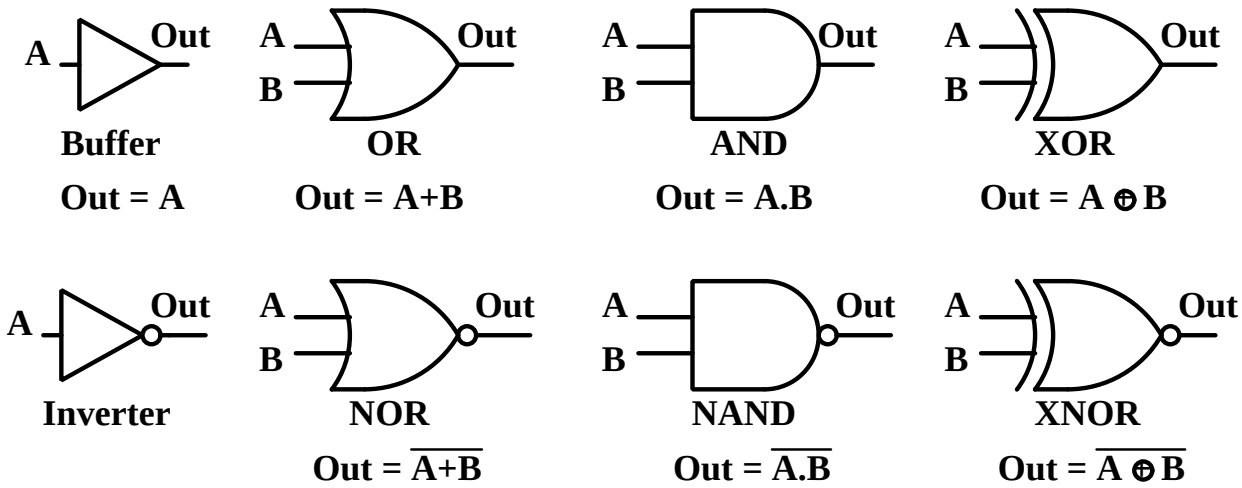


NAND output is pulled to ground when **both** inputs are '1'.
It is pulled up to V_{DD} when **either or both** inputs are '0'.

NOR output is pulled to ground when **either or both** inputs are '1'.
It is pulled up to V_{DD} when **both** inputs are '0'.

Logic Gates and symbols

Irrespective of the technology used for implementing the logic, standard symbols are used to represent logic functions.
The term “Logic Gate” is used for each logic function implemented in hardware. The figure below shows the symbols used for different types of logic gates.



If you leave an input to a TTL gate unconnected, it is taken as a ‘1’
Unused inputs to CMOS gates must **never** be left unconnected - this may result in heavy current flow and may damage the IC.

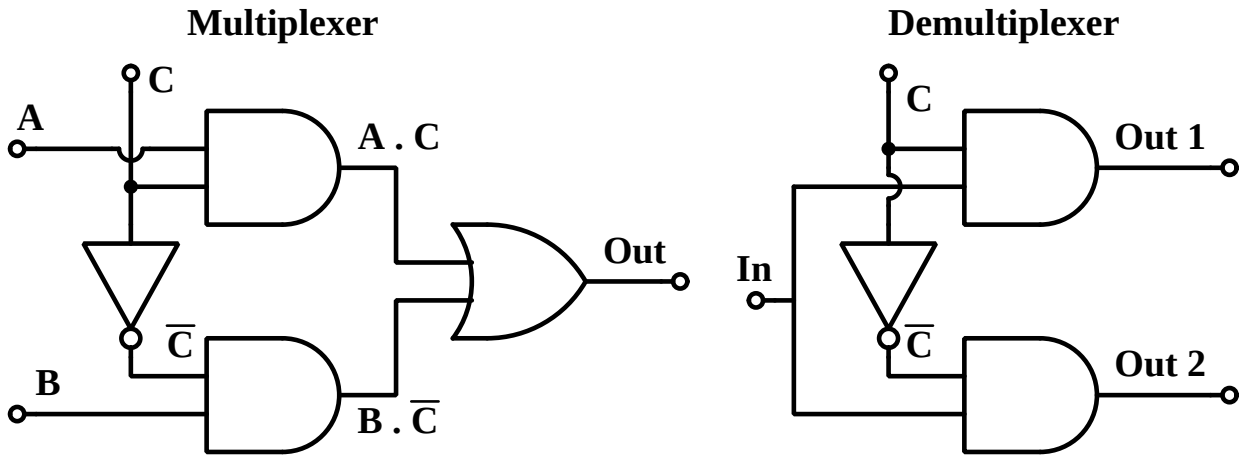
Some commonly used logic gates

Type numbers of TTL gates typically begin with 74 which is followed by a sub type of TTL like low power Schottky (LS) or High speed (H) and a number denoting the logic function.
CMOS B series has type numbers like CD 4xxx, while the TTL compatible CMOS logic gates have type numbers like 74Cxx.

Logic Function	No of gates on chip	TTL implementation	CMOS B Series
Inverter	6	7404	4069
2 input AND	4	7408	4012
2 input NAND	4	7400	4011
4 input AND	2	7421	4082
4 input NAND	2	7420	4012
2 input OR	4	7432	4071
2 input NOR	4	7402	4001
2 input XOR	4	7486	4070

Multiplexing and Demultiplexing

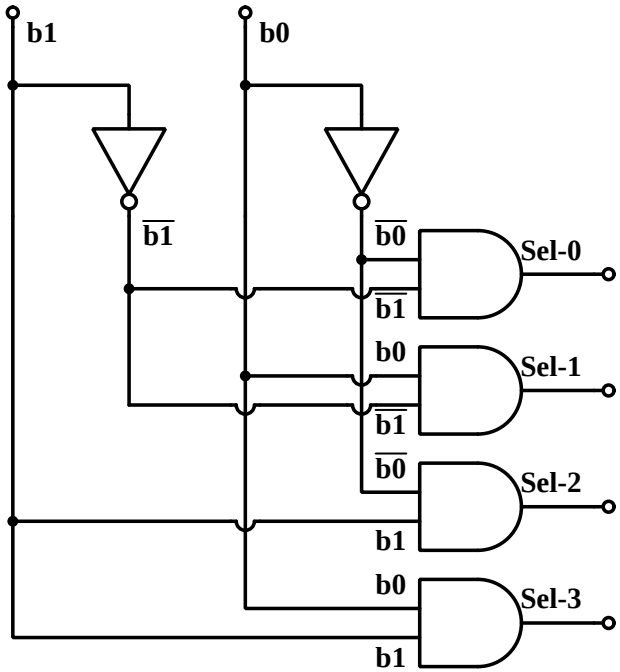
- We often want to select one out of two bits A and B depending on the value of C being '1' or '0'. The function $C \cdot A + \overline{C} \cdot B$ evaluates to A when C = '1' and to B when C = '0'.
- This can be used to select one out of two bits and put the selected bit on the output line. Circuits of this kind are called **multiplexers**.



- **Demultiplexers** do the opposite. The input data comes on a single line and depending on a line number code, it is put on one out of multiple output lines.

Decoding

What can we do if we want to multiplex from or demultiplex to more than two lines?
– We use multi-bit **decoders** which provide select lines to replace C and $\overline{C}$ in the previous example.

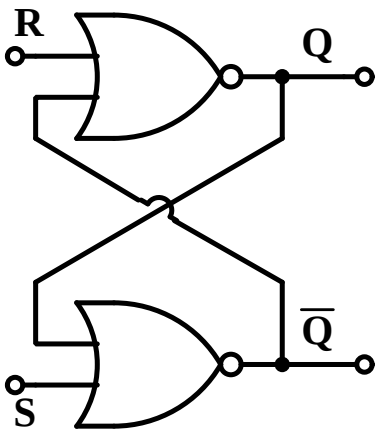


The circuit on the left shows a 2 bit decoder which provides 4 Select lines. The select signals replace C and $\overline{C}$ when we have more than 2 lines to multiplex or demultiplex.

Apart from their use in multiplexers and demultiplexers, these are widely used for address decoding and enabling selected devices connected to processors etc.

Storage Elements: RS Latch

Digital circuits require storage elements in addition to combinational logic functions. Consider the cross connected NOR gates in the circuit shown below.



For R = '0' and S = '0', the circuit can remain in one of these two states:

1. Q = '0' and $\overline{Q}$ = '1', (reset state)
2. Q = '1' and $\overline{Q}$ = '0'. (set state)

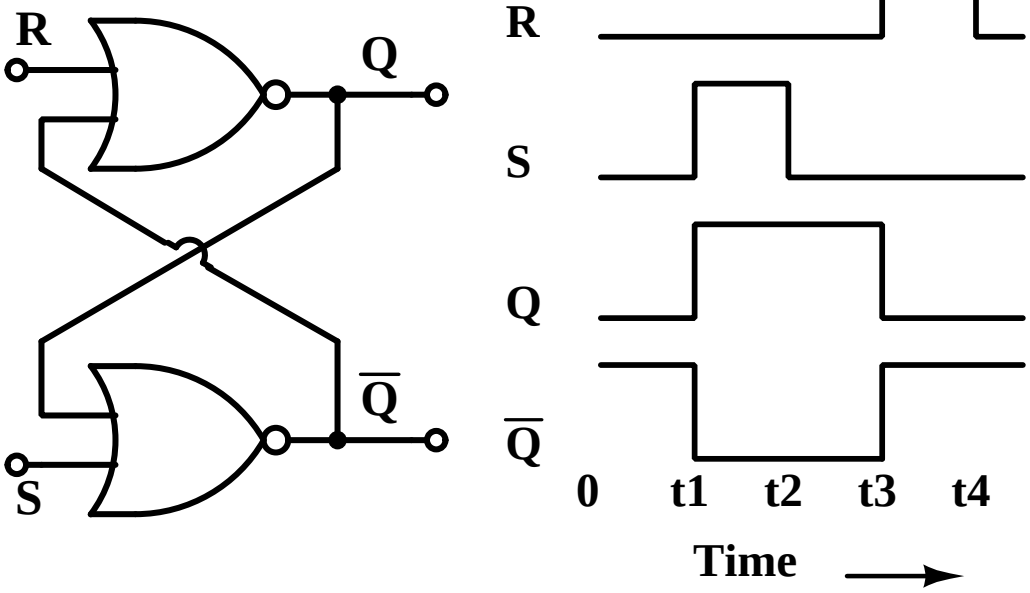
It is easy to see that for $R = 0$, $S \rightarrow$ '1' will force the circuit into its 'set' state (Q = 1, $\overline{Q}$ ='0').

$$S \rightarrow \text{'1'} \Rightarrow \overline{Q} = \text{'0'}, \quad R = \text{'0'}, \ \overline{Q} = \text{'0'} \Rightarrow Q = \text{'1'}$$

Now even if S returns to '0', the circuit will remain in its set state. Thus this circuit "remembers" that S had been '1' in the past.

Similarly, R = '1' while S = '0' will force the circuit in its 'reset state' and it will remain in reset state even after R returns to '0'.

RS Latch: set and reset action



Initially R = S = '0' and the latch is in reset (Q = '0', $\overline{Q}$ = '1') state.

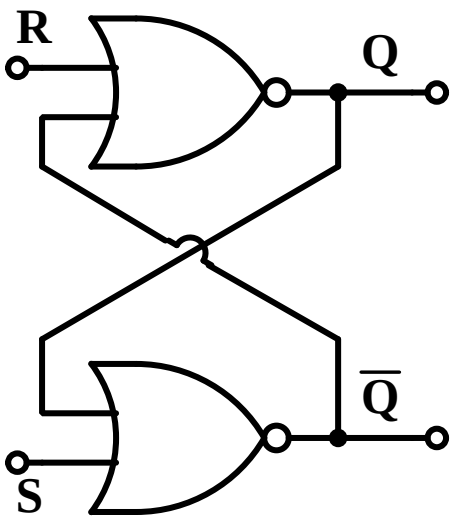
At t1, S goes to '1' and it forces $\overline{Q}$ to '0' unconditionally. Since R = '0' and $\overline{Q}$ = '0', Q goes to '1'.

At t2, S returns to '0', and the latch remains in set (Q = '1', $\overline{Q}$ = '0') state.

At t3, R goes to '1'. It forces Q to '0' unconditionally. Since S = '0' and Q = '0', $\overline{Q}$ goes to '1'.

at t4, R returns to '0'. With R = 0, S = 0, the latch remains in reset (Q = '0', $\overline{Q}$ = '1') state.

RS Latch: set and reset action



Since the S input sets Q to '1', it is called the 'Set' input.
R input returns Q to '0' and is termed as the 'Reset' input.
The latch retains its set (Q = '1', $\overline{Q}$ = '0') state even after S returns to '0'. Thus, it "remembers" that S had gone to '1' before it returned to '0'.

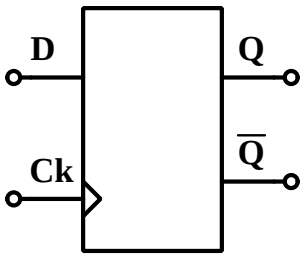
Similarly, the latch retains its reset (Q = '0', $\overline{Q}$ = '1') state even after R returns to '0'. Thus it "remembers" that R had gone to '1' before it returned to '0'.

If Set and Reset are simultaneously applied, both Q and $\overline{Q}$ are forced to '0'.
The latch will go to set or reset state when R and S return to '0', depending on which input is returned to '0' last.

Other storage circuits: D flipflop

There are other useful circuits which show a ‘memory’.

Many of these are used in sequential circuits with a clock.
Their action occurs whenever the clock has a specified transition, say from ‘0’ to ‘1’. This transition is known as the **active edge of the clock**.

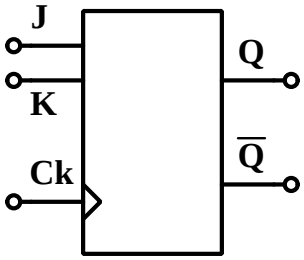


D flipflop: This is circuit with a ‘D’ (Data) input and a clock. At the active clock edge, it copies the value of D to its Q output and the complement appears at $\overline{Q}$.

The symbol for a D flipflop is shown above. The little triangle at the clock input shows that the flipflop transfers D to Q on the rising edge of the clock. A little circle is placed with the triangular symbol if the active edge is the falling edge of the clock.

Many circuit diagrams omit the indication for the active edge.

Other circuits with memory: JK flipflop



JK flipflop:
This circuit has two inputs J and K –
These act similarly to S and R inputs in the RS latch.

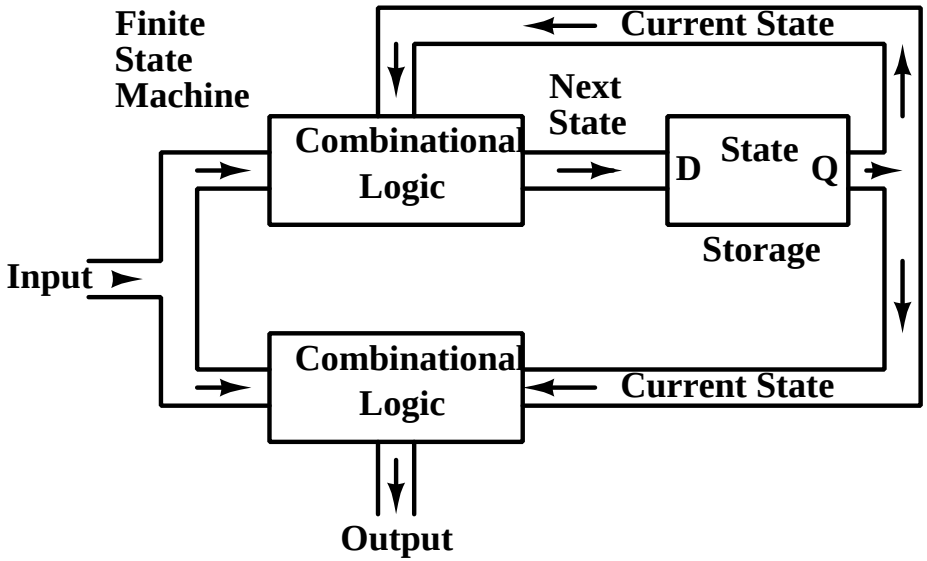
At the active edge of the clock,
If J = '0' and K = '0', the flipflop outputs remain unchanged.
If J = '1' and K = '0', Q goes to '1' and $\overline{Q}$ goes to '0'.
If J = '0' and K = '1', Q goes to '0' and $\overline{Q}$ goes to '1'.
If J = '1' and K = '1', it complements the values at Q and $\overline{Q}$.
(This and the presence of clock distinguish it from RS latch).

Toggle flipflop: In this, the J and K inputs are tied together and are used as the T input.
When T = '0', the flipflop outputs Q and $\overline{Q}$ remain unchanged.
When T = '1', it toggles *ie* complements the values at Q and $\overline{Q}$ at the active clock edge.

Finite State Machines

In a sequential circuit, the output sequence is a function of input sequence.
The sequences between which we want to establish a functional relationship have to be of finite length to be implementable.

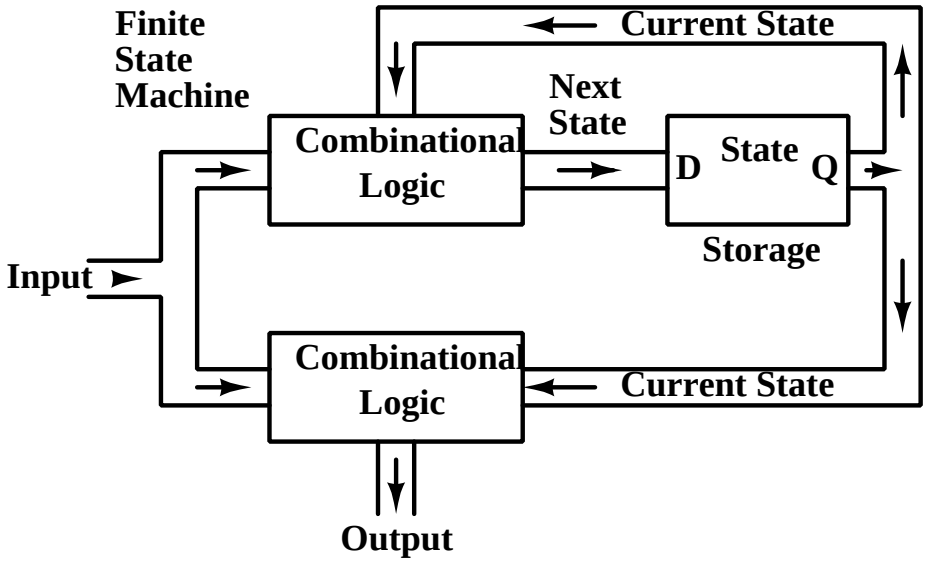
- A new set of input values are presented during each new clock period.
- It is not necessary to store all the past values of input variables.



A (possibly multi-bit) **state** is stored, which is a digital value with sufficient information about the past inputs so that the current element of the output sequence can be generated as a **combinational** function of the state and the current inputs.

Finite State Machines

- In each clock period, as fresh inputs arrive, a new value of **state** is generated which is a combinational function of the current value of state and current inputs.
- The state is thus a recursive function of previous inputs.

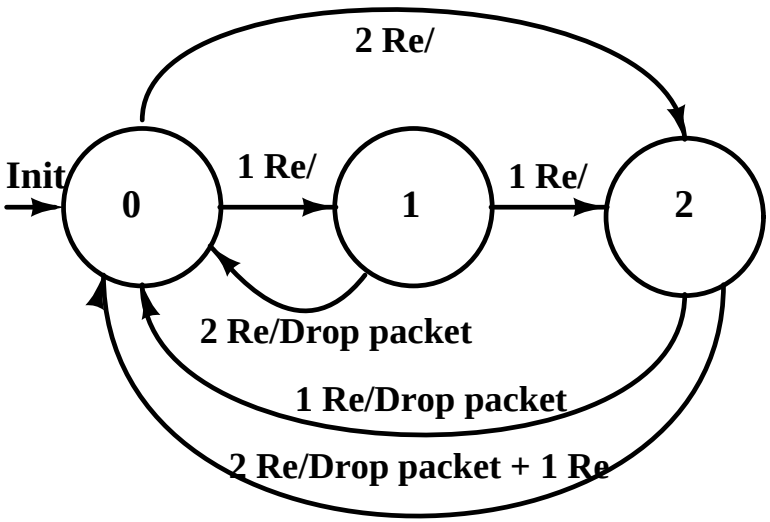


The **state** represents the "history" of input variables as relevant to production of the output sequence.

Systems which use this approach to generate sequential logic are called **Finite State Machines** or FSMs.

FSM example: Penny in the slot machine

Consider a Penny in the slot machine which accepts 1 Re and 2Re coins. It dispenses a packet worth 3 Rs. and returns coins if required.

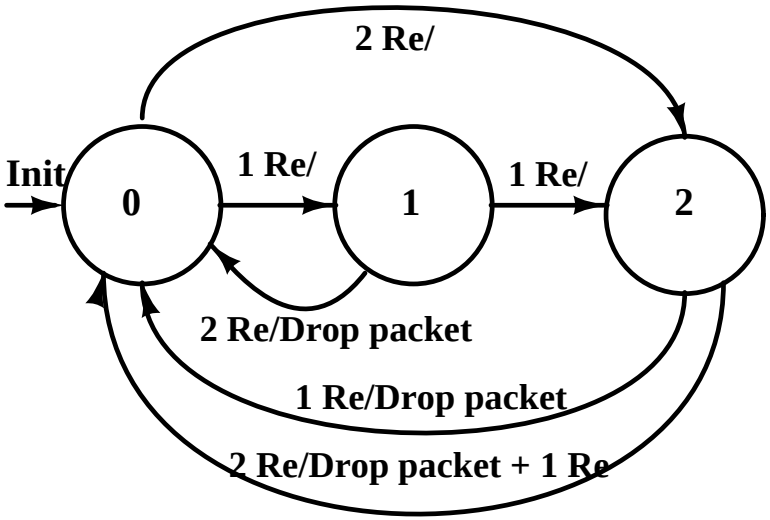


The state diagram on the left shows the evolution of 'states' in response to a sequence of inputs.

- At any time, one of two inputs may occur – insertion of a 1 Re coin or insertion of a 2 Re coin.
- In response, the machine may – i) do nothing, ii) Drop the packet or iii) Drop the packet and a 1 Re coin.

The machine can be designed with 3 states and combinational logic to determine the action and the next state in response to specific inputs.

FSM example: Penny in the slot machine



At power on, the machine wakes up in state 0.

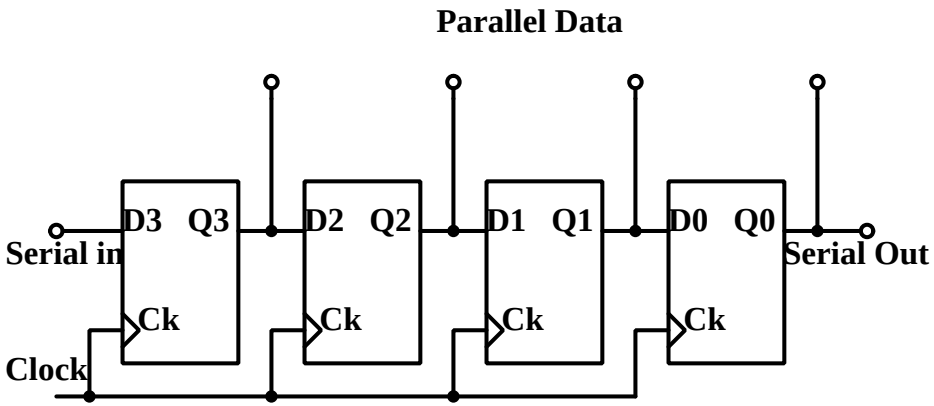
- At any time, The machine is in one of the three states shown.
- In any state, only two external inputs are possible: Arrival of a 1 Re coin or arrival of a 2 Re coin.

Curr st	Input	Output	Next St	Input	Output	Next St
0	1 Re	Nothing	1	2 Re	Nothing	2
1	1 Re	Nothing	2	2 Re	Drop packet	0
2	1 Re	Drop packet	0	2 Re	Drop packet + 1 Re	0

We can implement this with 2 flipflops to encode the state number and combinational logic to determine next state and output from current input and state.

Registers

- A group of D flipflops in parallel can be used to store a multi-bit value. This is called a **register**.
- We can also connect D flipflops in series as shown below. This arrangement is called a **Shift Register**.



At each active edge of the clock, $Q0 \rightarrow$ Serial Out, $Q1 \rightarrow Q0$, $Q2 \rightarrow Q1$, $Q3 \rightarrow Q2$, Serial In $\rightarrow Q3$.
Thus data is shifted to the right by one location.

- This can be used for serial to parallel conversion by feeding data in serially and after all data has been fed, collecting it in parallel.
- Shift register are also used for parallel to serial conversion. Data is loaded in parallel to all the D flipflops and shifted out serially.
- These are also used for data delay, as a serial memory etc.

Number Systems

- We have already seen how we can represent natural numbers in a base-2 system (binary representation).
  The bit pattern $b_{n-1} \ldots b_i \ldots b_1 b_0$ represents the number:
  $$N = \sum_{i=0}^{n-1} 2^i \cdot b_i.$$
- Given a decimal number N, how do we determine b_i?
  - Take N and divide by 2. The remainder is b_0.
  - Repeat this process by dividing the quotient by 2 and retaining the remainder as $b_1, b_2 \ldots$. Continue till the quotient becomes 1 or 0. This is then the most significant bit.
  - For somewhat larger numbers, it is more efficient to convert the number to base-16 or Hex format, by using the above procedure but dividing by 16 every time.
  - The resulting Hex number can be converted to binary easily by replacing each hex digit by its binary equivalent.

An n bit binary number can represent any natural number in the range $0 \leq N \leq 2^n - 1$.

Representation of Natural numbers

Let us illustrate the conversion procedure by a few examples.

Given $N = 55$.
Divide successively by 2.

$55 = 27 * 2 + 1$, so $b_0 = 1$
$27 = 13 * 2 + 1$, so $b_1 = 1$
$13 = 6 * 2 + 1$, so $b_2 = 1$
$6 = 3 * 2 + 0$, so $b_3 = 0$
$3 = 1 * 2 + 1$, so $b_4 = 1$
Quotient = 1, go $b_5 = 1$

Thus decimal $55 =$ binary 110111

Given $N = 1000$. Divide by 16.
$1000 = 62 * 16 + 8$,
so least significant hex digit is 8.
$62 = 3 * 16 + 14$,
so the next digit is 14 or E.
$3 < 16$, so the most significant digit is 3. Thus $1000 =$ 3E8 in Hex.
Expanding each hex digit to binary, this can be written as:

$$0011 \mid 1110 \mid 1000.$$

Binary addition

- Binary numbers can be added just like decimal numbers, with a carry being transferred to a more significant bit position whenever the sum exceeds 1.
- We can represent positive numbers between 0 and $2^n - 1$ using n bits.
- What happens if the sum exceeds the largest representable number?

Consider an example using 4 bit numbers. Using 4 bits we can represent numbers between 0 and 15

Now consider the addition of 11 to 7 in binary arithmetic using 4 bits. 1011 + 0111 = 10010. The overflowing 5th bit can be used to signal that overflow has occurred and the result is not valid.

However, this provides us with a clue to a method for representing negative numbers.

Representing Negative numbers

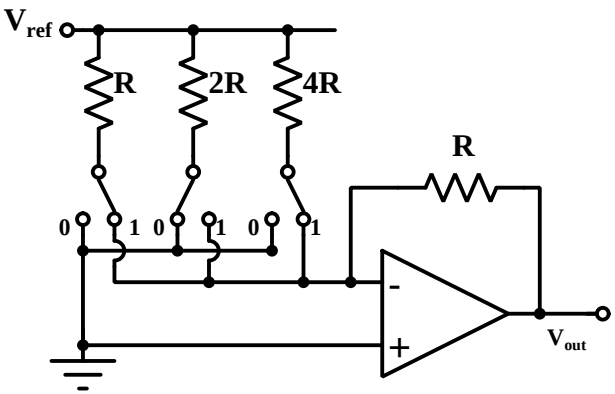
- In arithmetic, we define a negative number by the property: $x + (-x) = 0$.
- To represent negative numbers, we take the magnitude of the number and then choose a number which when added to it will give zeros in the defined bit width, (albeit with an overflow).
- For example, the decimal number 5 is 0101 using 4 bit representation. When we add decimal 11 to it we get 0101 + 1011 = 10000. (5+11=16). If we ignore the overflow (fifth bit), the result is zero.
- Thus, if x = 0101, (-x) = 1011. Now, x + (-x) = 0, if we ignore the overflow.
- Obviously, we cannot use 1011 to represent 11 as well as -5! So we adopt a convention where numbers with 0 in the most significant position are positive, while numbers with 1 in the most significant position represent negative numbers.

Representing Negative numbers

- We use the convention where numbers with 0 in the most significant position are positive, while numbers with 1 in the most significant position represent negative numbers.
- With this convention, signed 4 bit numbers can be represented in the range -8 to + 7, inclusive of Zero.
- How do we find the number which when added to x will give zero (with overflow)?
- Notice that when we add a number to its complement (changing all 0's to 1 and 1's to zero), the sum will have 1 in all positions. Now if add 1 to this, we'll get all zeros with an overflow.
- So to get the negative of a number, we take its complement (also called 1's complement) and add 1 to it. (This is called 2's complement).

Digital to Analog Conversion

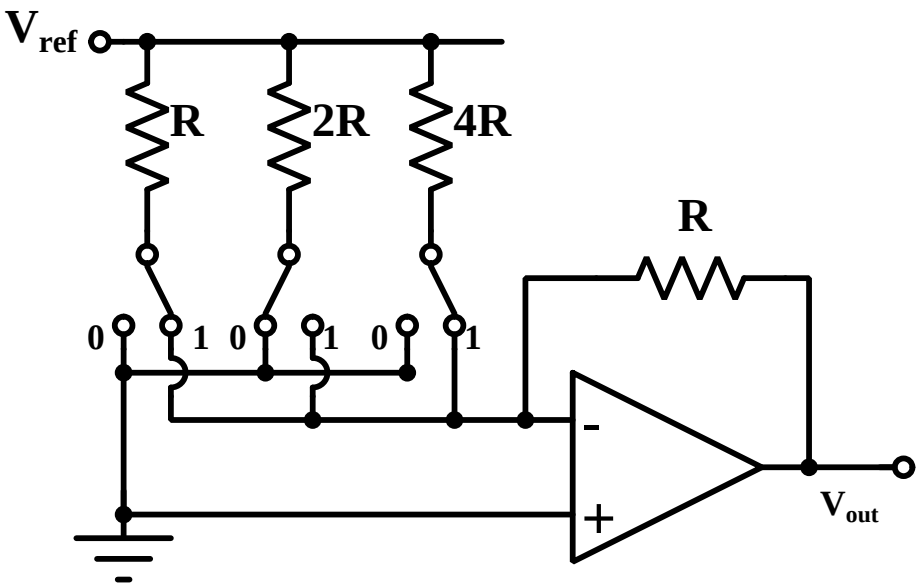
Consider the production of an analog quantity – say voltage, proportional to a given digital numbers. This is Digital to Analog conversion or DAC.



A digital number: $b_{n-1} \ldots b_i \ldots b_1 b_0$ represents $\sum_0^{n-1} 2^i b_i$.
If we use resistors with resistance values proportional to 2^i and use these to draw current from a reference voltage V_{ref} (with the other end of the resistors at ground potential), the currents will be in binary ratio.

We put a two way switch at the end other than V_{ref} which connects either to ground or to the virtual ground of an opamp.
Since these two choices are at the same potential, the current remains unchanged if we switch it between either of these.

Digital to Analog Conversion



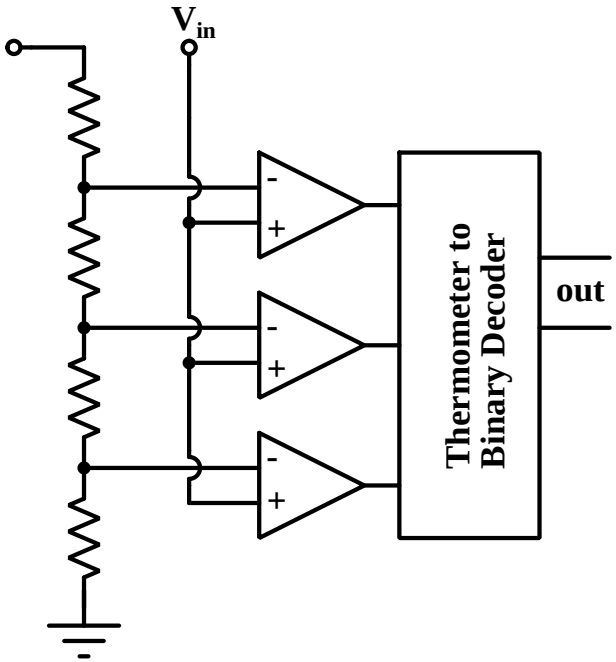
Each two way switch is controlled by a bit of the digital number which we want to convert. Since the two way switches direct the current either to ground or to the virtual ground of an opamp, the current through the resistors remains unchanged if we switch it between either of these.

The current going into the inverting terminal is then proportional to the digital number. The output voltage is $-R$ times this current and thus proportional to the digital number.

This method of D to A conversion is intuitive, but getting accurate values of resistors over a wide range is a difficult task.

Analog to Digital Conversion

A flash ADC is a fast converter which uses a separate comparator for each digital value possible. A resistor divider creates equally spaced N-1 reference levels.



- A bank of comparators compares the input to each of the reference levels.
- All comparators with reference level below the input value go to '1' while all those above it go to '0'.
  This is known as **Thermometer coding**.
- A decoder is then used to transform the thermometer code to binary.

Flash converters are used for A to D conversion with a small number of bits, since its complexity grows exponentially with each additional bit.

Analog to Digital Conversion

- Many other types of A to D converters are available, providing a compromise between speed, power and complexity.
- A successive approximation ADC uses a DAC whose Digital input is provided by a register. The ADC sequentially adjusts the bits of this register and compares the output of the DAC with the input voltage, till they match. This type of ADC is widely used in medium speed applications.
- A dual slope ADC provides low complexity at low conversion rates. It converts its input voltage to a time interval, which is measured using a counter. This kind of ADC is used by most DMMs.

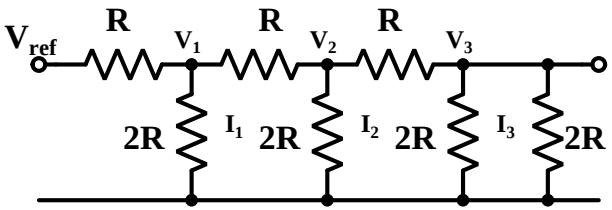
From the user's view point, a "start conversion" signal is sent to the ADC (of whatever type) and when the conversion is complete, it sends an "end of conversion" signal. The output of the ADC can be read as a digital number once the conversion is complete.

And That's all

for now!

(for Digital Design).

DAC with R-2R ladder network



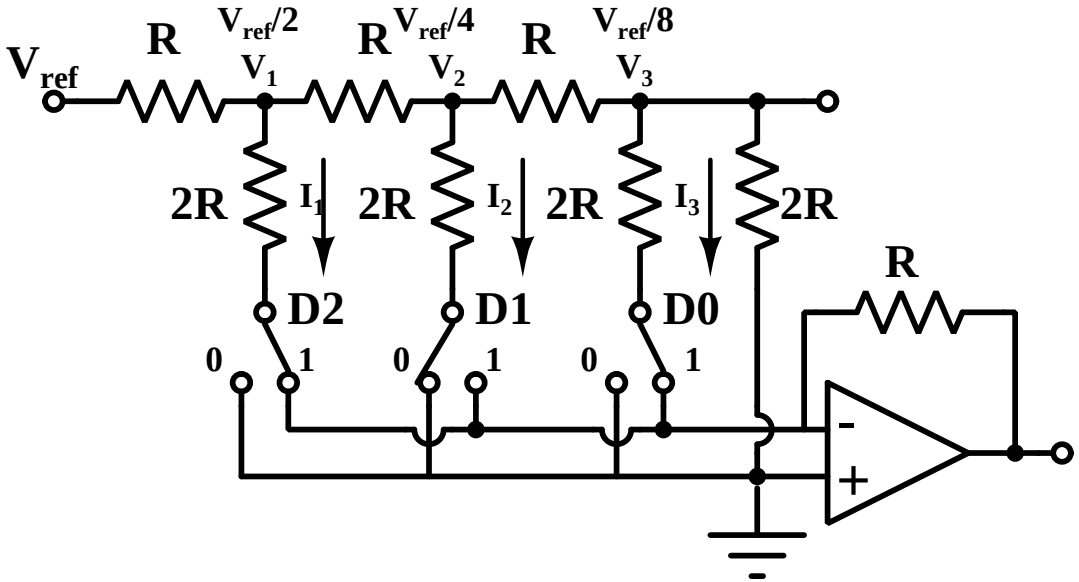
Consider a network with resistors of value R in series and those of value 2R in parallel in a ladder network as shown on the left.

Notice the extra 2R resistor at the end of the chain.

- At the end, we have 2 resistors to ground, each of value 2R. This is equivalent to a single resistor of value R.
- This equivalent resistance of R and the series resistance R from V_2 form a potential divider such that $V_3 = V_2/2$.
- The series resistor from V_2 and the equivalent resistor R to ground provide a total resistance of 2R to ground from V_2.
- Extending this argument, $V_2 = V_1/2$ and $V_1 = V_{ref}/2$
- Correspondingly, the current to ground through the 2R resistors is in binary ratio.
- by inserting a two way switch at the ground end and switching between ground and virtual ground, we can generate a voltage proportional to the digital value.

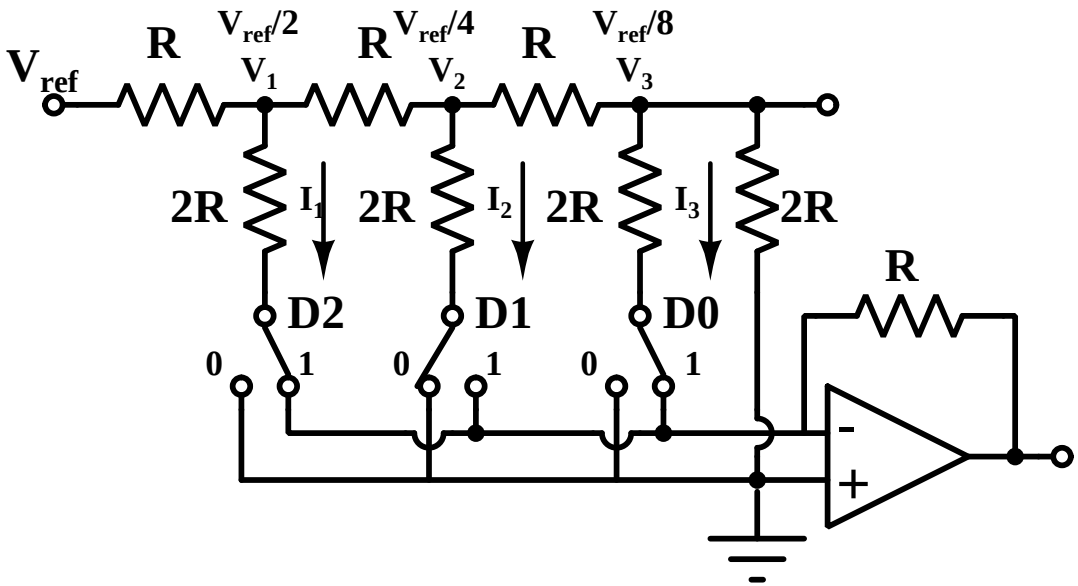
DAC with R-2R ladder network

The Circuit below shows a D to A converter using the R-2R ladder network.



- Bits of the digital value choose whether the current through the 2R resistors will go to ground or to virtual ground.
- Total current going into the virtual ground is converted to a voltage using a feedback resistor.

DAC with R-2R ladder network



- Implementing the DAC is easier this way, because it requires only two resistance values. In fact, two identical resistors are put in series to get the 2R value.
- Accurate matching of resistors is much easier here compared to the binary weighted resistor case.

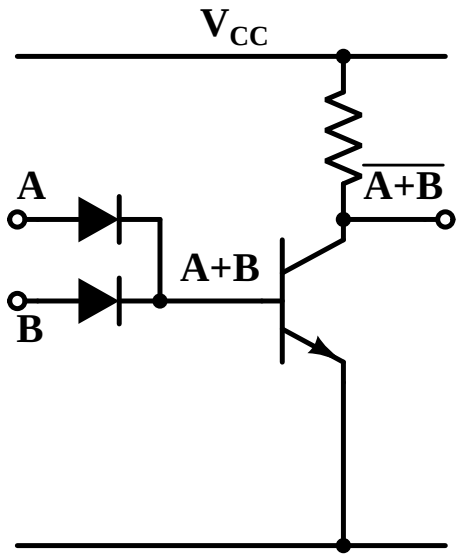
ADC with successive approximation

- This is a widely used architecture for ADCs.
- It uses a DAC and a comparator.
- At the first step, the most significant bit for the DAC is set.
- The comparator indicates if the unknown voltage is above or below the DAC output. If the unknown value is below the DAC value, the bit which was last set is cleared.
- Now the next significant bit for the DAC input is set. The unknown voltage is compared to the DAC output again.
- This process is continued till the least significant bit has been determined.

Logic families: DTL

Earliest implementation of Digital logic used relays connected in series/parallel to implement logic.

With the advent of semiconductor devices, diodes and transistors were used to implement digital logic. The circuit on the right is a DTL NOR gate.
A HIGH voltage (close to V_{CC}) represents a logic '1', while a LOW voltage (close to ground) represents a logic '0'.



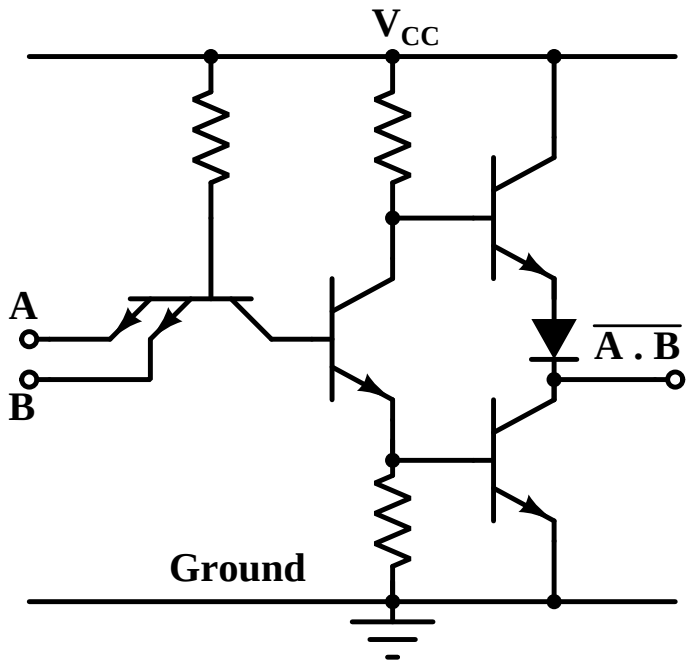
When either A or B (or both) are HIGH, base current flows, saturating the bipolar transistor. This pulls the output LOW.
The transistor is OFF when both A and B are LOW as there is no base current. The output is then connected to V_{CC} through the collector resistor and is HIGH.

Logic families: TTL

Diode-Transistor Logic was eventually replaced by Transistor-Transistor Logic or TTL.

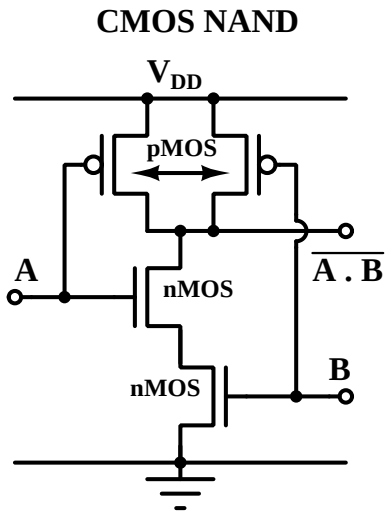
This was the dominant digital technology for implementation of digital logic for a long time.

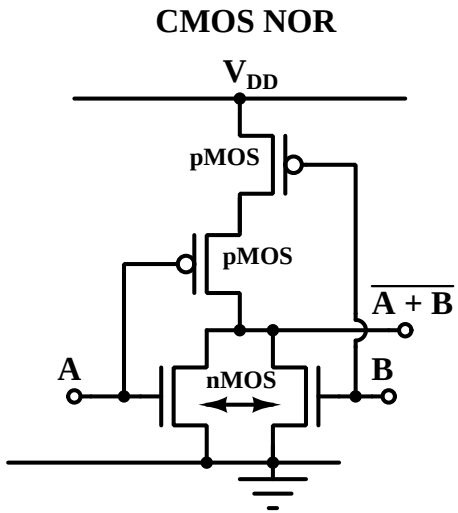
- TTL replaces the diodes at the input in DTL by a bipolar transistor with multiple emitters.
- The circuit on the right shows a NAND gate implemented in TTL technology.
- The output stage is called a “totem pole” circuit.



Logic families: CMOS

- Modern digital circuits use MOS transistors.
- A ‘1’ at the gate input turns the nMOS ON and the pMOS OFF
- A ‘0’ at the gate input turns the nMOS OFF and the pMOS ON.



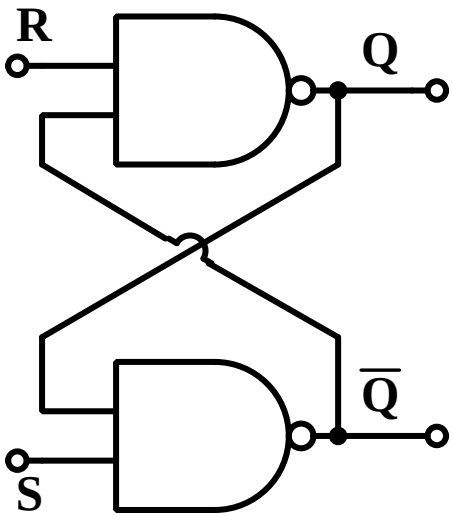


NAND output is pulled to ground when **both** inputs are ‘1’. It is pulled up to V_{DD} when **either or both** inputs are ‘0’.
NOR output is pulled to ground when **either or both** inputs are ‘1’. It is pulled up to V_{DD} when **both** inputs are ‘0’.

By using complementary MOS transistor types (n channel and p channel), high speed can be obtained at reasonably low power consumption. This is known as CMOS logic.

RS Latch with cross connected NAND gates

Remember the duality property of Boolean logic?



We can also construct an RS latch using cross connected NAND gates. Its operation is similar to the cross connected NOR.

Here the circuit is idle when R and S inputs are at '1', while the 'set' and 'reset' action is triggered by S or R going to '0'.

Like the cross connected NOR circuit, Set and Reset are not supposed to go to their active level ('0' in this case) simultaneously.

If both R and S inputs go to their active level ('0') simultaneously, Q as well as $\overline{Q}$ will go to '1'.

The output will settle to one of the two stable states depending on which of the two inputs is removed from the active state last.

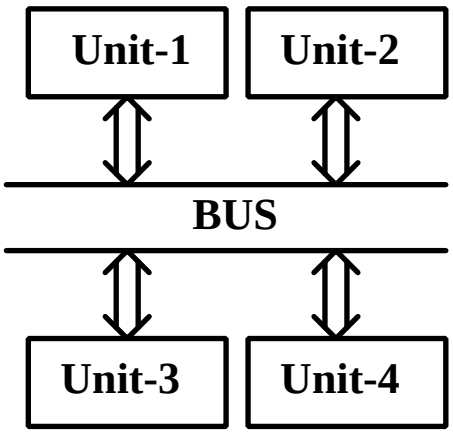
Introduction to Microprocessors

Dinesh Sharma
D. Chakraborty, K. Chatterjee, B.G. Fernandes, J. John,
P.C. Pandey, N.S. Shiradkar, K.R. Tuckley

Department of Electrical Engineering
Indian Institute of Technology, Bombay

January 24, 2024

Connecting Digital Circuits using a “Bus”



For Data communication between multiple units with digital outputs, it is convenient if we can connect their input/output ports using common wires. This is called a **bus**.

However, outputs of conventional digital circuits cannot be shorted together.

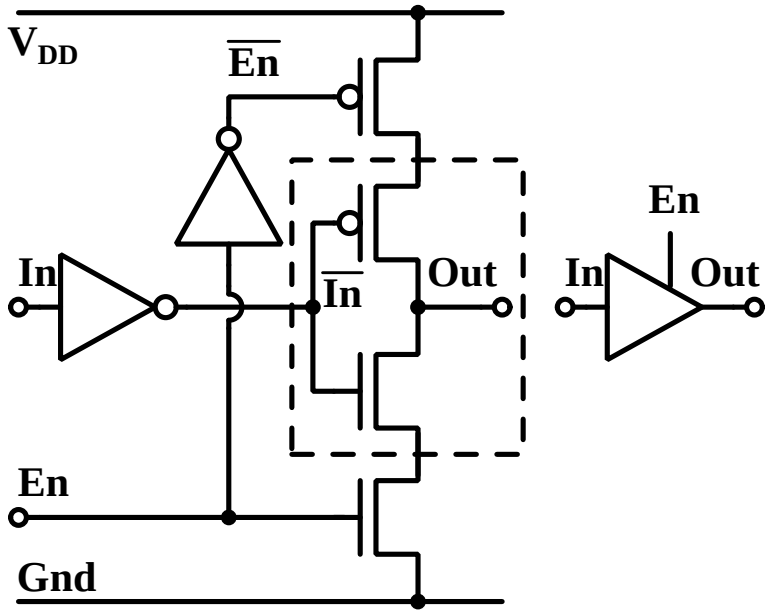
- Outputs of conventional digital circuits can only be ‘0’ or ‘1’. If we short the outputs of multiple circuits, some will try to make the output ‘High’ while the others may try to pull it ‘Low’.
- This can lead to indeterminate outputs and heavy current being drawn from the supply.
- To connect outputs of multiple circuits to the same wire in the bus, we need circuits with **tri-stateable** outputs or **open-collector outputs**.

Tri-stateable Devices

- The output of tri-stateable digital circuits can have three states. It could be ‘0’ or ‘1’ like the conventional digital circuits, or it could be in a high impedance state called ‘Z’.
- In the high impedance state, the circuit is **disconnected** from the output and does not interfere with other outputs connected to the same wire.

The figure on the right shows the circuit diagram and the symbol for a tri-stateable buffer.

The circuit has an extra input to enable it (– labelled as ‘En’ in the circuit).
When En = 1, the circuit acts as a buffer, but when En = 0, the circuit is disconnected from the output.



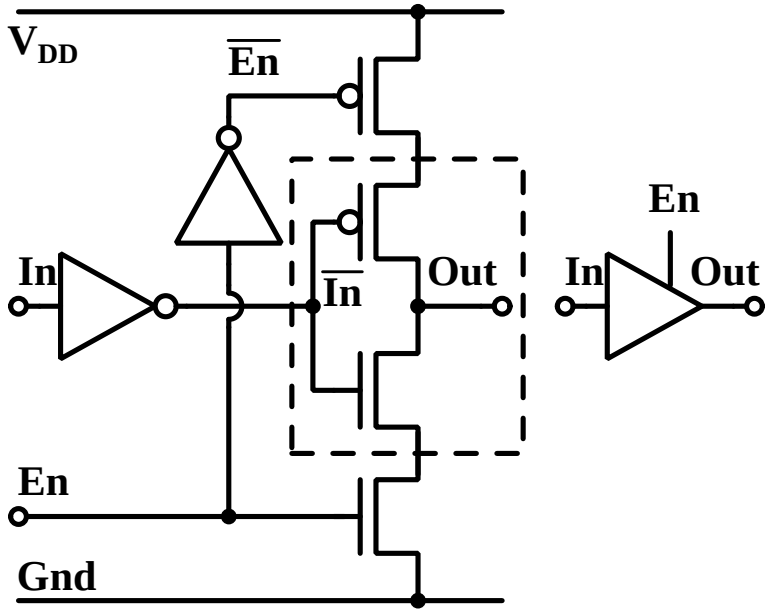
Tri-stateable Buffer Circuit

Circuits which can disconnect their output drivers from the output terminal are called **tri-stateable** circuits.

The connection to supply and ground is through the top pMOS and bottom nMOS transistors.

Both of these are OFF when En = 0 (and therefore $\overline{\text{En}} = 1$).

Both are ON when En = 1 and $\overline{\text{En}} = 0$, applying power to the circuit in the dashed box.



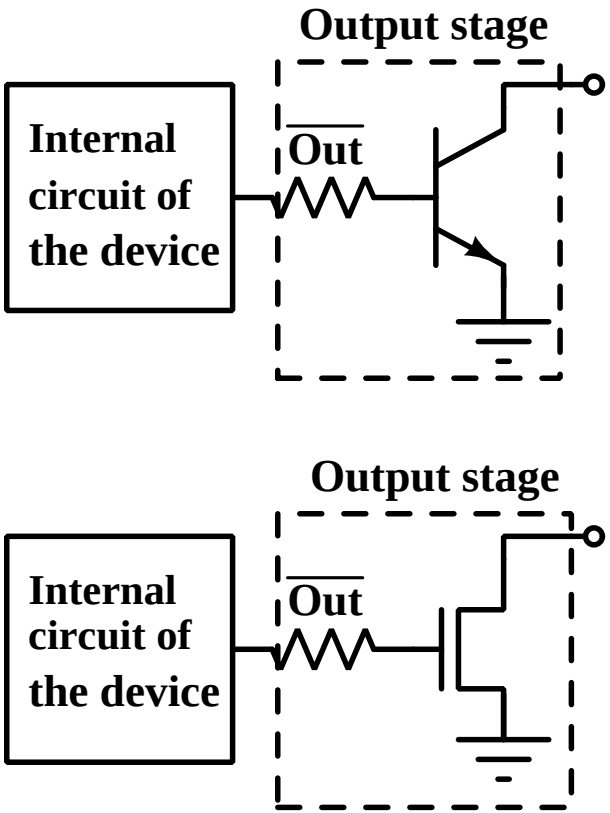
When En = 1, the middle two transistors in the dashed box form an inverter, which inverts the output of the first inverter, thus providing a buffer function.

Bus Control with Tri-stateable devices

- One of the digital circuits on the bus acts as the **“bus master”**.
- It decides which of the circuits on the bus will be **listeners** for the data on the bus and which will be the **talker**.
- While there can be multiple listeners on the bus, only one circuit can be a talker. The bus master manages this by setting the Enable line of the designated talker to TRUE and that of all others to FALSE.
- The bus itself contains many wires for carrying the data and a few additional wires driven by the bus master for bus administration functions such as designating roles as listeners or talkers.

Open Collector Outputs

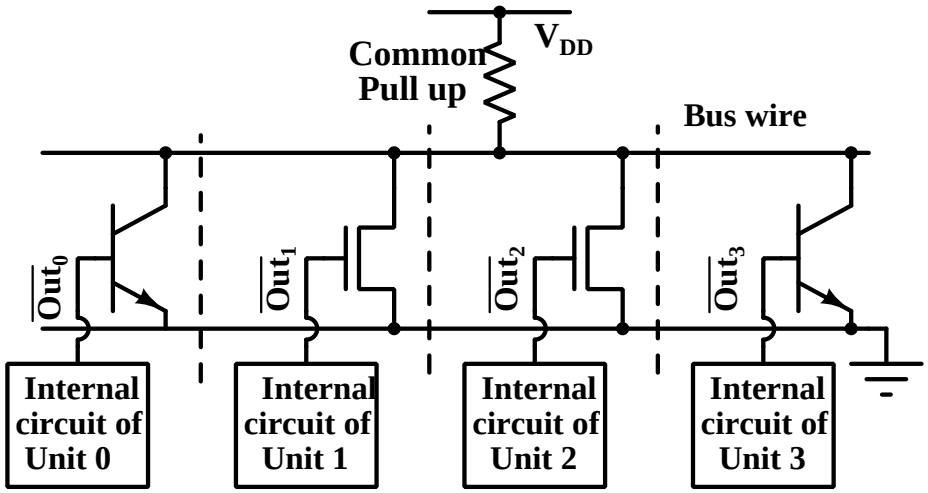
Another way for direct connection of outputs to common wires is to use open-collector/open-drain output stages.



- The output stage of digital circuits normally includes a switching device for pulling the output up to the supply voltage, as well as a device to pull it down to ground.
- However, open collector/open drain circuits include only the pull down device.
- The collector/drain of the pull down device is brought out as the output terminal.

Outputs of open-collector/open-drain circuits can be shorted together. A common external resistor is used as a pull up for the shorted outputs.

Open Collector Outputs for Bus connection



- There is a common pull up resistor for all output devices which are effectively in parallel.
- Units turn OFF their output device to write a ‘1’ to the output and turn it ON to write a ‘0’ to the output.

- It is possible for different units to output different logic values.
- The bus wire will be pulled up by the common pull up resistor when **all** devices output a ‘1’ (turning their output device OFF).
- The bus wire will be pulled down to ‘0’ if **any** device outputs a ‘0’ (turning its output device ON).
- A ‘0’ is dominant over a ‘1’ in this system. A device writing a ‘1’ to the output is effectively disabling itself.

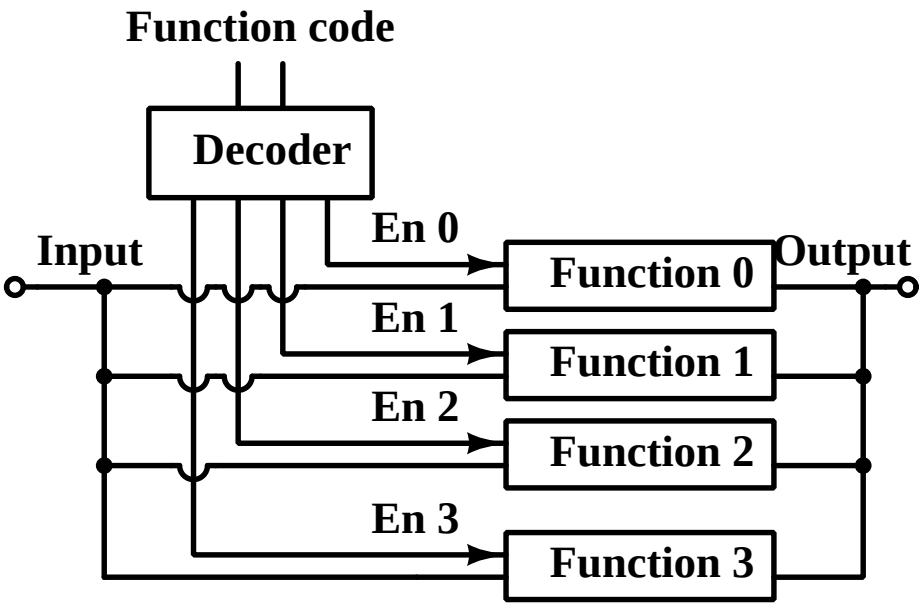
Need For a programmable device

- Design of complex integrated circuits has a high fixed cost component.
- If sold in large quantities, the cost of an integrated circuit can be made quite low.
- It makes sense to design a programmable device which can be configured to perform different functions for different products.
- Now this device can be used for a variety of products and will thus be required in large quantities.
- The specific function performed by the device needs to be selected by a digital input. This is called an **instruction**.
- The device is given a series of instructions to perform a task. This sequence of instructions is a **program**.

The Processor

- The programmable digital device which can run a program is called a processor.
- A processor which is implemented on a single chip using VLSI technology is called a **microprocessor**.
- Modern microprocessors can be quite complex – a Pentium processor used in PCs has upwards of 100 million transistors!
- The basic logic in a processor just performs the following loop endlessly:
  - Fetch next instruction
  - Decode it
  - Execute the instruction

Notional Plan for a Programmable Device



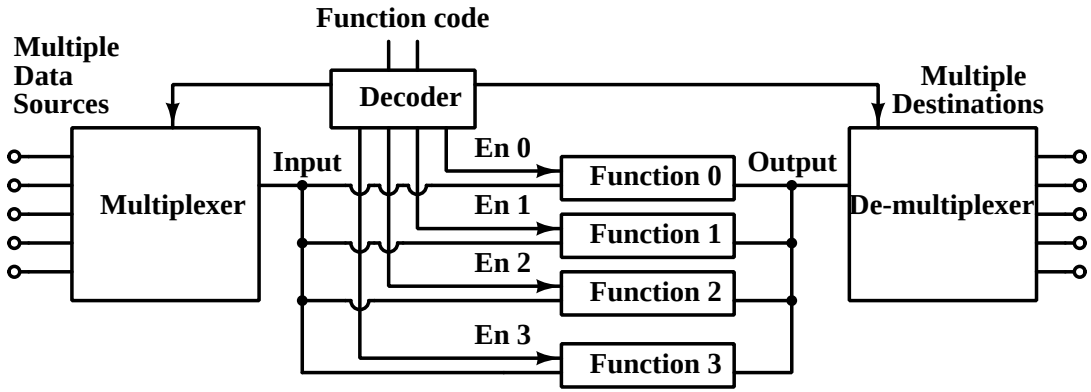
- A notional plan for making a programmable device is shown on the left.
- The common input going to the inputs of all functional devices is fine, but the common output will cause problems if made from conventional digital circuits!

We can use tri-stateable outputs to share a common output terminal.

However, each tri-stateable circuit needs an individual enable signal. These enable signals are contained in the function code and are provided by the decoder.

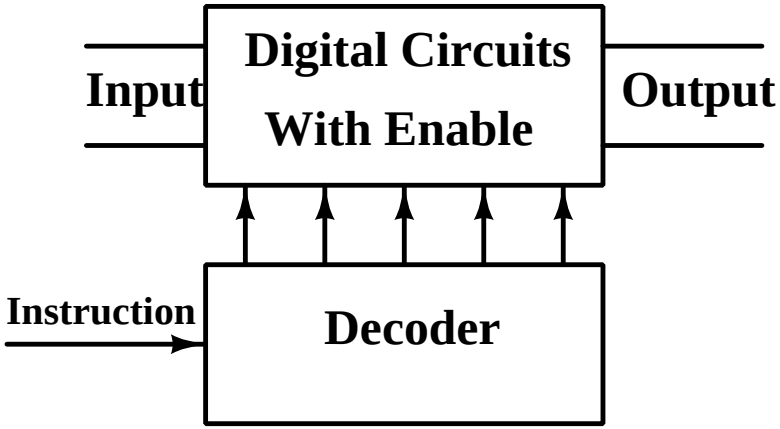
A multipurpose Digital Circuit

Assume that we have multiple digital circuits, each performing a different function.



- A specific circuit can be selected using its enable input.
- We need a multiplexer to route the **data** from multiple sources to the input of this circuit and a de-multiplexer to route its outputs to a selected device.
- The multiplexer and demultiplexer will also need ‘select’ inputs.
- All this information has to be supplied to configure this circuit to perform a selected function.

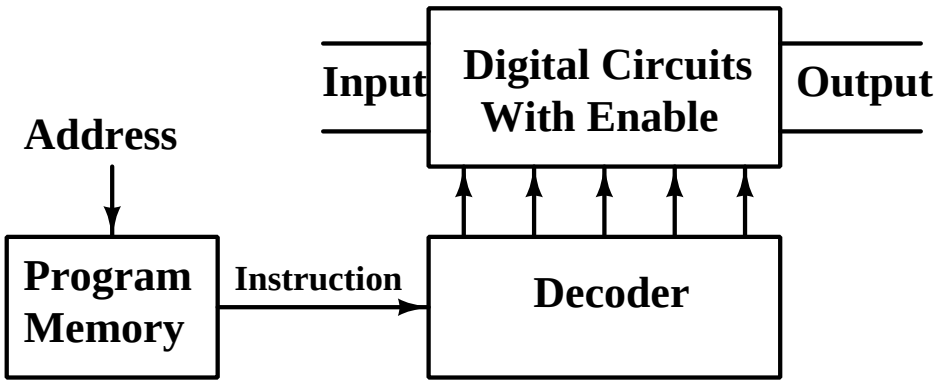
Choosing a specific sub-circuit



A lot of control inputs will be required to choose and configure a specific circuit and to route the data from input through this circuit to the output.

- We can encode this information in a compact form and use a decoder to generate all the detailed control signals necessary for data routing and for enabling circuits.
- This encoded information is called an **instruction**.
- The format of this information is decided by the circuit designer and the decoder expands out the control signal according to this format.

Storing Instructions

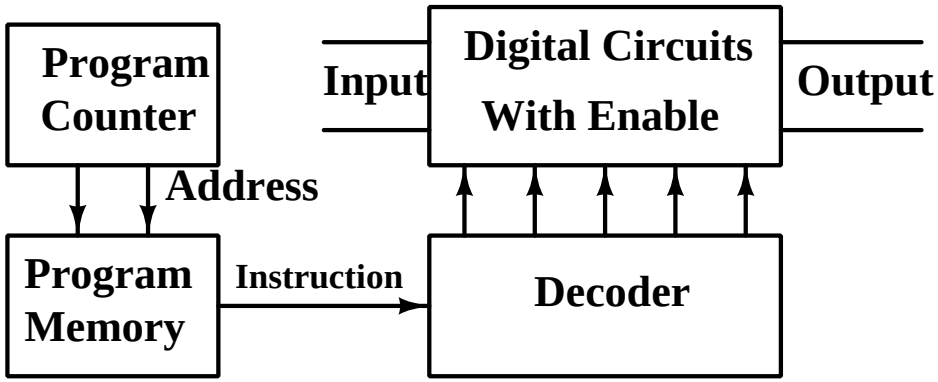


- The actual task performed by this circuit may require a series of operations to be carried out.
- For example, we may wish to multiply a variable by a coefficient and then add the product to an accumulating sum.

- This requires a series of instructions, to be executed in a sequence.
- This sequence of instructions is called a **program**.

We store the sequence of instructions in a memory called the *program memory*.

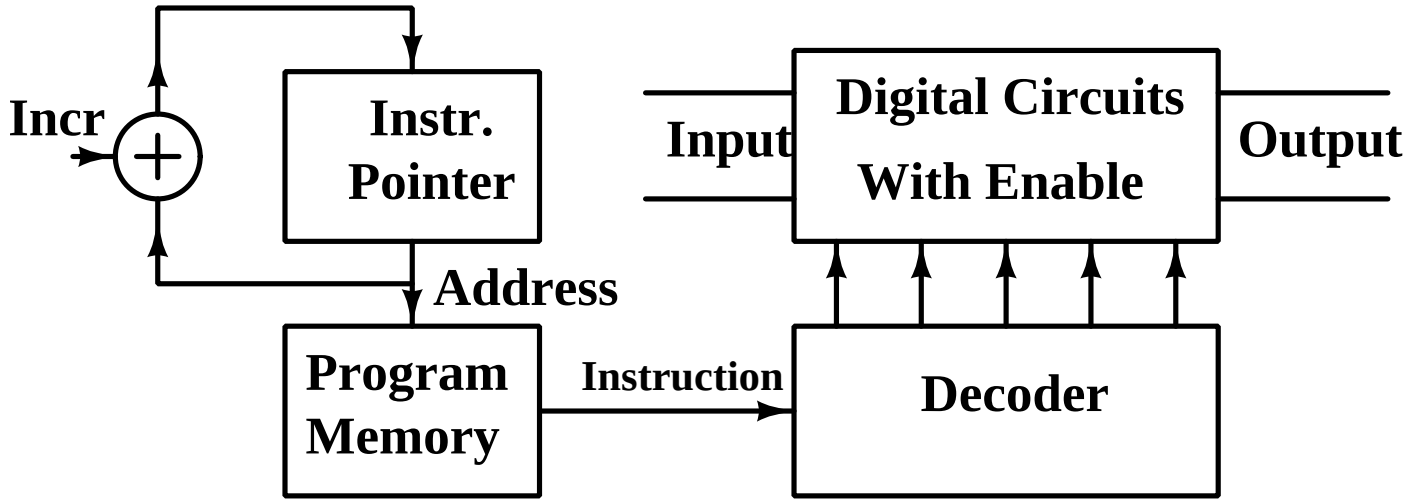
Fetching Instructions



- To fetch each instruction, we need to provide its address to the memory.
- From where will this address come?

- We can include a counter to generate sequential addresses for the program memory.
- Every time an instruction is fetched from the memory, the counter will increment the address to point to the next instruction.
- The counter which provides addresses to the program memory is called a **program counter**.

Instructions From Auto-incremented Addresses

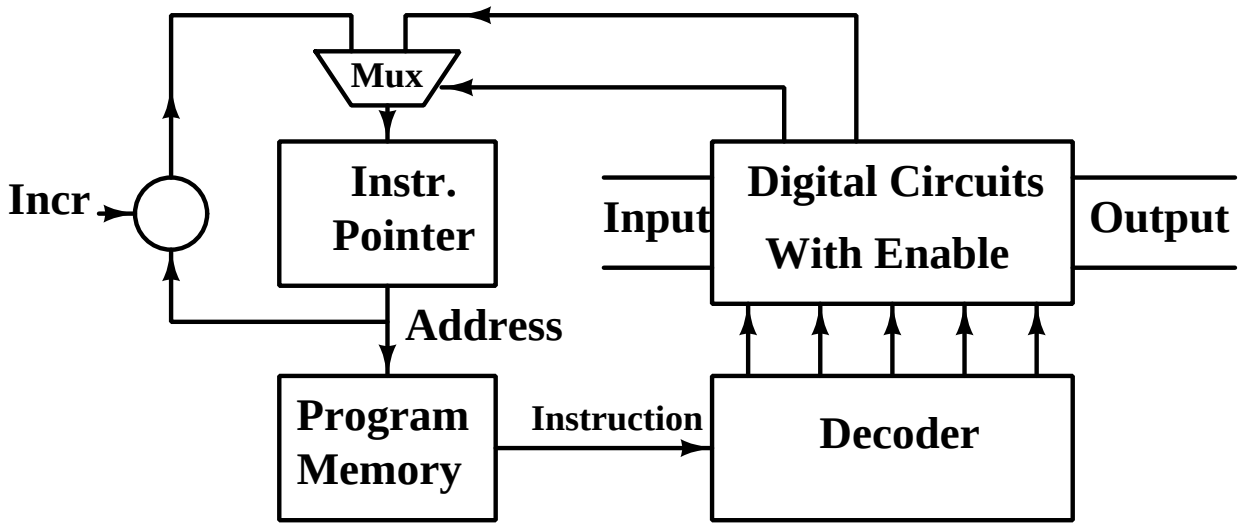


Fetch - Decode - Execute

- The program counter is just a register to store a value and an adder to increment the value stored in the register.
- This register is also referred to as the **instruction pointer**.

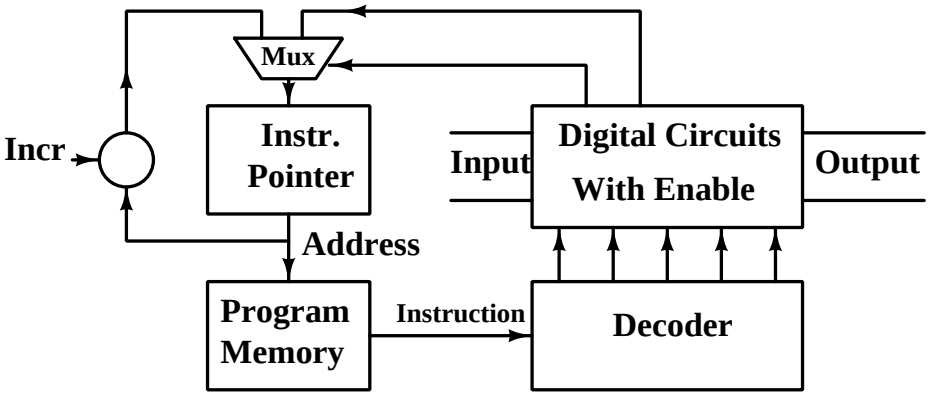
Instructions from non sequential addresses: “Jump”

The instruction pointer needs to be initialized to some value when the whole operation starts.



- Instructions need not be executed in a strict sequence. After finishing a block of instructions, we may wish to execute a different block of instructions, stored at some other address.
- The ability to load data into the instruction pointer gives us the ability to alter the flow of instructions in the program during its run time.
- When a new value is loaded in the instruction pointer, the next instruction will be fetched from this new location.

Executing a “Jump” instruction

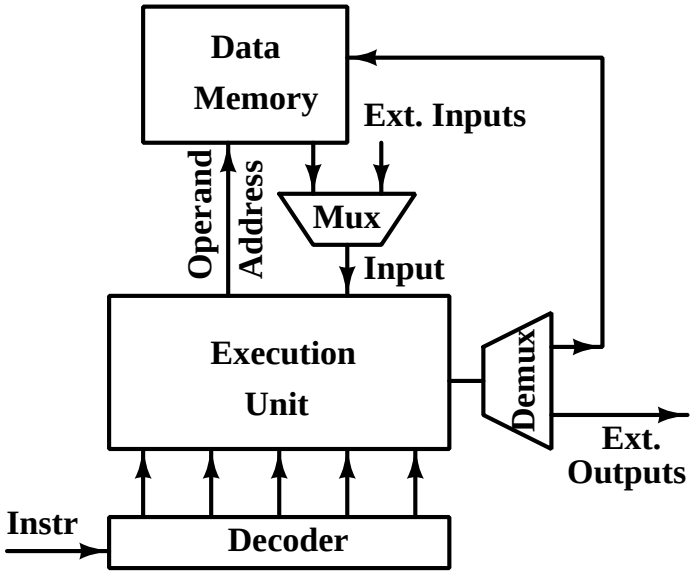


After finishing a block of instructions, we may wish to “jump” to a different block of instructions, stored at some other address.

- Moving to a different block of instructions may be accomplished by loading a different value into the instruction pointer.
- The value to be loaded into the instruction pointer can be provided by the last instruction of the first block.
- execution of this instruction will cause the multiplexer to choose this new address, rather than the default incremented address as the input to the instruction pointer.

Managing data

So far we have concentrated on the flow of instructions and control path. What about the data path?



Different instructions will, in general, operate on different data.

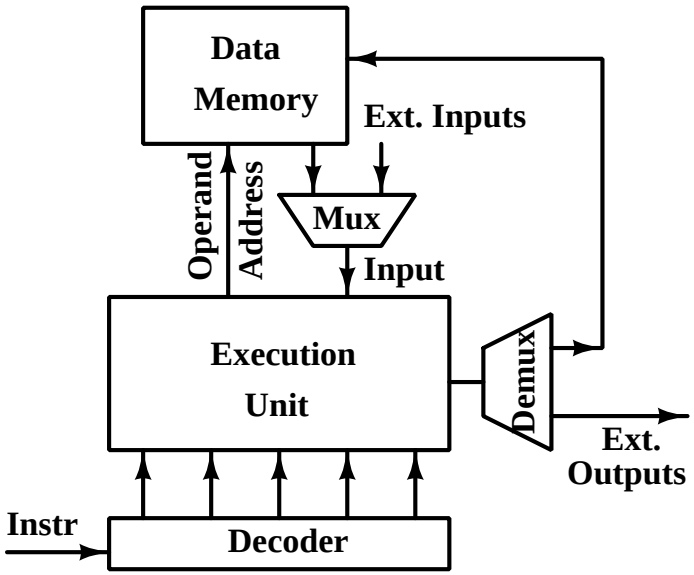
How do we propose to manage the flow of data?

$\rightarrow$ Data can also be stored in a memory. Successive data items can then be fetched from/written to this memory.

- We shall need the address where the data item is to be fetched from/written to.
- The actual address of the data item is often stored in a register. (This register is called a data pointer).
- The instruction should identify the register which stores the address.

Data I/O with external world

The circuit we are designing should also be capable of fetching data from external sources and to output data to destinations other than the data memory.

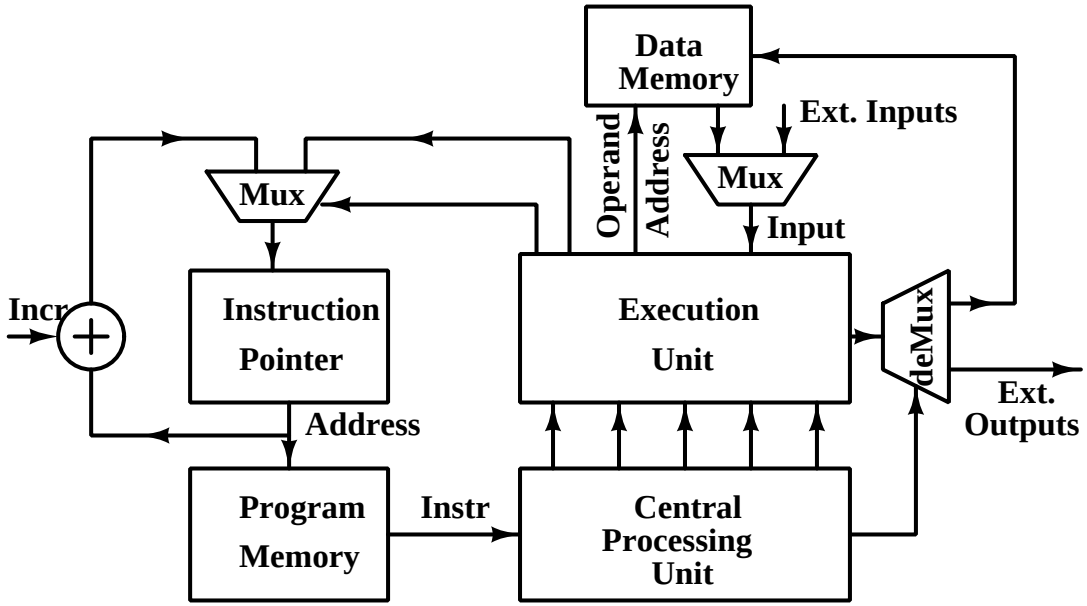


- We can place a multiplexer in the input data path and a demultiplexer in the output data path in order to choose between data memory and external IO.
- Control for these multiplexers/demultiplexers should also be included in the instruction.

- The data and program memory need not be physically distinct.
- Different architectures choose either a common memory and data path for both instructions and data, or separate data and instruction memories with independent paths.

The Central Processing Unit

We have oversimplified the evolution of this circuit by assuming that just a combinational circuit like a decoder is needed to execute instructions.



Fetch - Decode - Execute

- In fact each instruction may need a sequence of actions to be executed properly.
- Therefore, we need a sequential circuit like a finite state machine to interpret and carry out an instruction.

- The circuit which generates the sequence of control inputs to the execution unit is called the Central Processing Unit or CPU.

The Central Processing Unit

- The central processing unit generates control inputs for the sequence of operations to be carried out.
- At the top level, it repetitively carries out the three steps:
  - Fetch the next instruction
  - Decode the instruction
  - Execute it
- Each of these operations may involve a series of transactions with the memory or with internal execution units.
- Instruction execution may involve data processing or controlling the flow of instruction through writing to the instruction pointer.

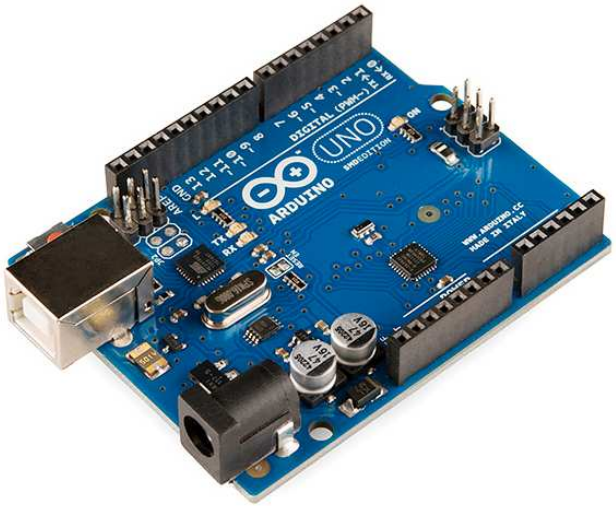
We have designed a microprocessor! How to use it?

- The architecture which has evolved through this discussion describes the skeleton of a microprocessor.
- However, we need many devices external to the microprocessor to make a useful system – For example, program and data memories, display device drivers, keyboard readers etc.
- So with microprocessors, we invariably need multiple ICs on a circuit board to perform any useful function.
  For example, the old 8085 processor needed a program memory chip (say, type 2764, 8Kx8 EEPROM), data memory chip (type 6264, 8Kx8 RAM), 3 to 8 decoder (74LS138) to select I-O devices, data transceiver etc. in order to make an operational circuit.

Microcontroller based systems

- As technology has progressed, it has become possible to put many of the components required in the system along with the microprocessor on the same IC.
- Processors which have these components on the same IC are called **microcontrollers**.
- A system using microcontrollers can be quite compact and can be made operational in practical systems with very few external components.
- Ready-made cards with microcontrollers and the essential external components are available with varying capabilities.
- “Raspberry Pi” is an example for such cards and uses the popular ARM processor. This is a 32 bit or 64 bit system and is used in somewhat high end applications.

Microcontroller based systems



- “Arduino” is a family of cards whose design is open source. These are therefore made by several manufacturers and are quite economical.
- There are many models of Arduino cards. The most popular among these are UNO R3 and nano, which are based on the 8 bit AVR microcontroller ATMega 328P.
- These cards are suitable for low-cost and simple applications. The UNO R3 card can be bought for under Rs. 500, while the nano card is even cheaper.

Microcontroller ATMega 328P

- Arduino R3 and nano boards use the microcontroller ATMega 328P.
- This microcontroller contains:
  - an 8 bit microprocessor with 32 registers,
  - 32KB of flash memory to store the program,
  - 1 KB Electrically Erasable and Programmable memory,
  - 2 KB SRAM to store data,
  - 3 Counter/Timers,
  - 10 bit A to D converter with 6 input channels,
  - Serial communication using RS232C, I^2C as as well as SPI protocols.

The Arduino **board** adds a USB controller to the microcontroller resources to provide USB connectivity.

Application Development with Microcontroller boards

Having selected a microcontroller board with the necessary resources, we need to program it to perform the function we need. As discussed earlier, this can be done using instructions recognized by the microcontroller.

- Instruction to the microcontroller are groups of ‘1’s and ‘0’s – which do not make much sense to human beings. These instructions are said to be in **machine code**.
- Meaningful mnemonics are associated with each machine code instruction which indicate the purpose for that instruction.
  For example, we associate MOV R31, R0 with a pattern like: 0010 1101 1111 0000 for an instruction which copies the contents of register number 0 to Register no. 31.
- The collection of such mnemonics are then “assembled” into machine code by a program (typically run on an external computer).
- This program is called an **assembler**.

Application Development with Microcontroller boards

- While assembly language programs make sense to human beings, these are very detailed and it is laborious to write programs in assembly language.
- High level languages (such as C and C++) can express what needs to be done much more compactly.
- It is then possible to translate this high level language program to assembly language and eventually to machine language using a program called a **compiler**.
- Once the translation is complete, the bit pattern corresponding to the machine language must be downloaded to the microcontroller. This requires a communication program to run on the external computer and a receiver program to run on the microcontroller.
- The receiver program is a small program which is pre-loaded into the program memory of the microcontroller at the time of manufacture. This is called **boot code**.

Application Development with Microcontroller boards

The application development cycle involves

1. Writing the application program in a high level language (For example, C or C++).
2. Compiling it on the PC – usually using an Integrated Development Environment (IDE) supporting the board we shall be using. (The Integrated Development Environment includes an editor for writing the program, a compiler, a library with useful pre-written functions and the software required to send the machine code to the microcontroller board).
3. Downloading the compiled program to the board using IDE on the PC and the boot loader on the microcomputer.
4. Running the downloaded program on the board after resetting the processor.
5. Testing and debugging the application program.

Still to come . . .

In the next lecture, we'll discuss the application development process with a few examples.

Application Development with Microcontrollers

Dinesh Sharma
D. Chakraborty, K. Chatterjee, B.G. Fernandes, J. John,
P.C. Pandey, N.S. Shiradkar, K.R. Tuckley

Department of Electrical Engineering
Indian Institute of Technology, Bombay

January 24, 2024

Structure of a Traditional Program

- Traditional computer programs start with some initialization code which gives starting values to variables and configures devices as desired.
- The next task is computation of desired values. The program asks for inputs as and when it needs them and may read values from external devices which are connected to the system.
- When outputs are ready, it provides these in the form of displays, prints, plots or outputs to external devices.
- The circuit simulation program “ngspice” is an example of such a program.
- The computation algorithm decides when inputs will be taken, when output will be provided etc.
- When all the outputs have been provided, the program terminates.

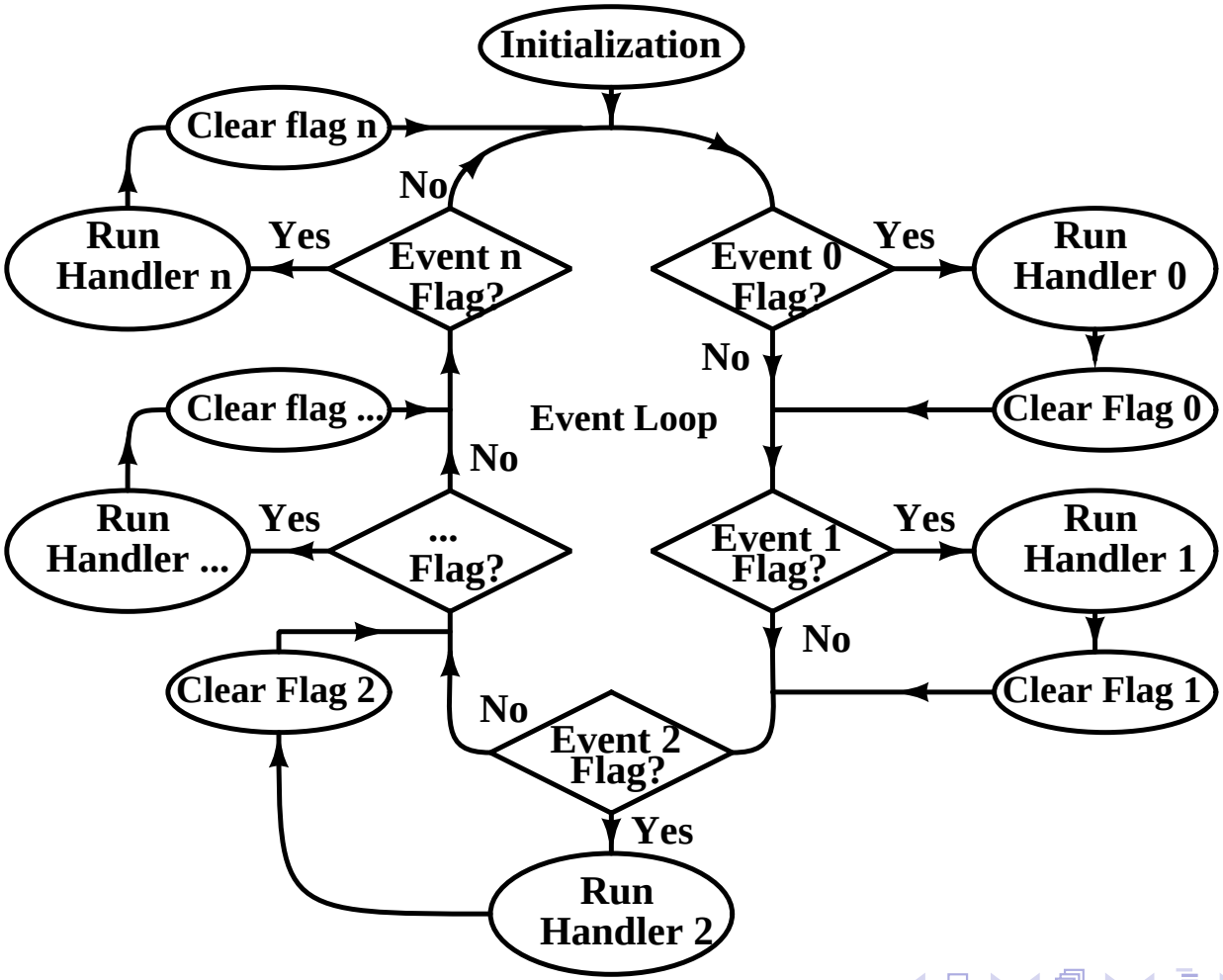
Structure of an Event Driven Program

The structure of an event driven microcomputer program is quite different.

- Microcomputers are typically used in embedded systems.
- These are applications where the presence of a microprocessor or a program is not apparent to the user.
  For example, a digital camera has a microcomputer which is running a program, but the user is not aware of this. The interface presented to the user is that of a camera.
- These applications are event driven. External events decide which portion of the program will run at any time.
- These programs never terminate ...
  What would happen to a camera if its program terminated?
- So the basic program structure is that of an initialization phase followed by an endless loop.

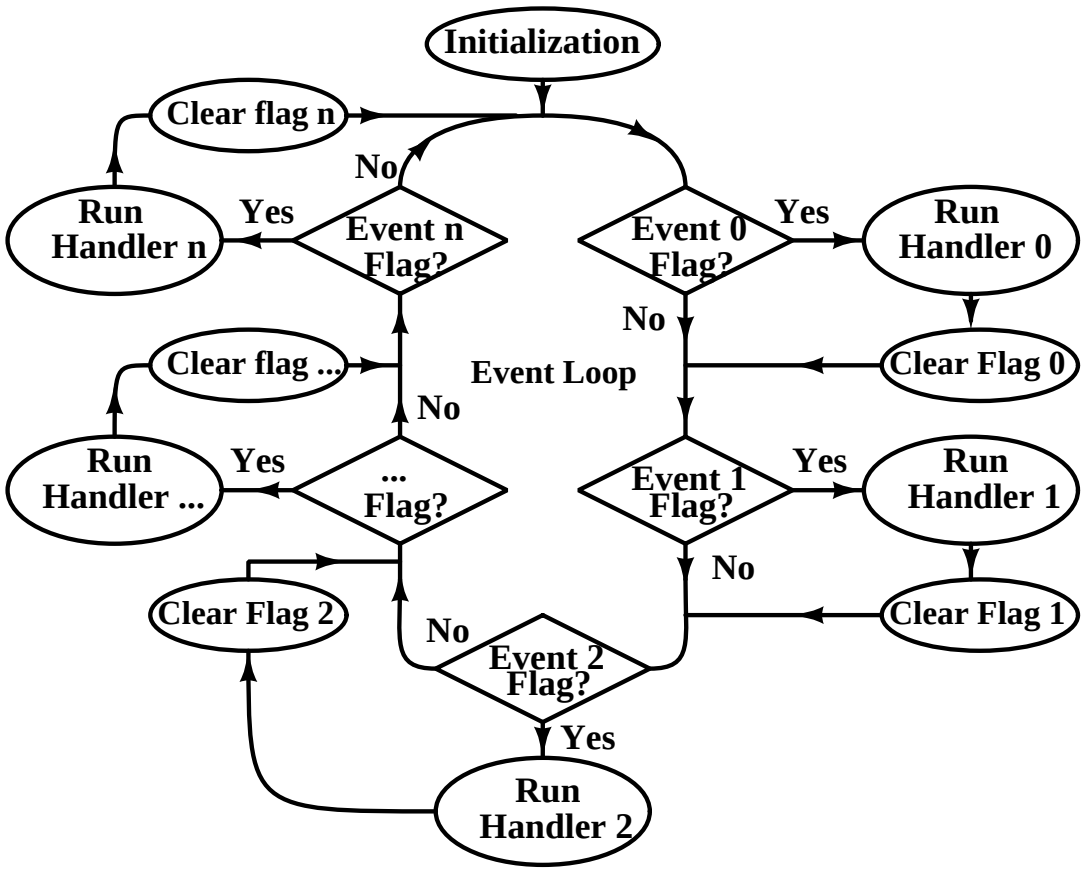
Structure of an Event Driven Program

An event-driven program enters an endless loop after the initialization phase.



Structure of an Event Driven Program

The program is aware of multiple events which might occur in the external world. It includes software (called an event handler) for every such event. This software should run whenever the event occurs.

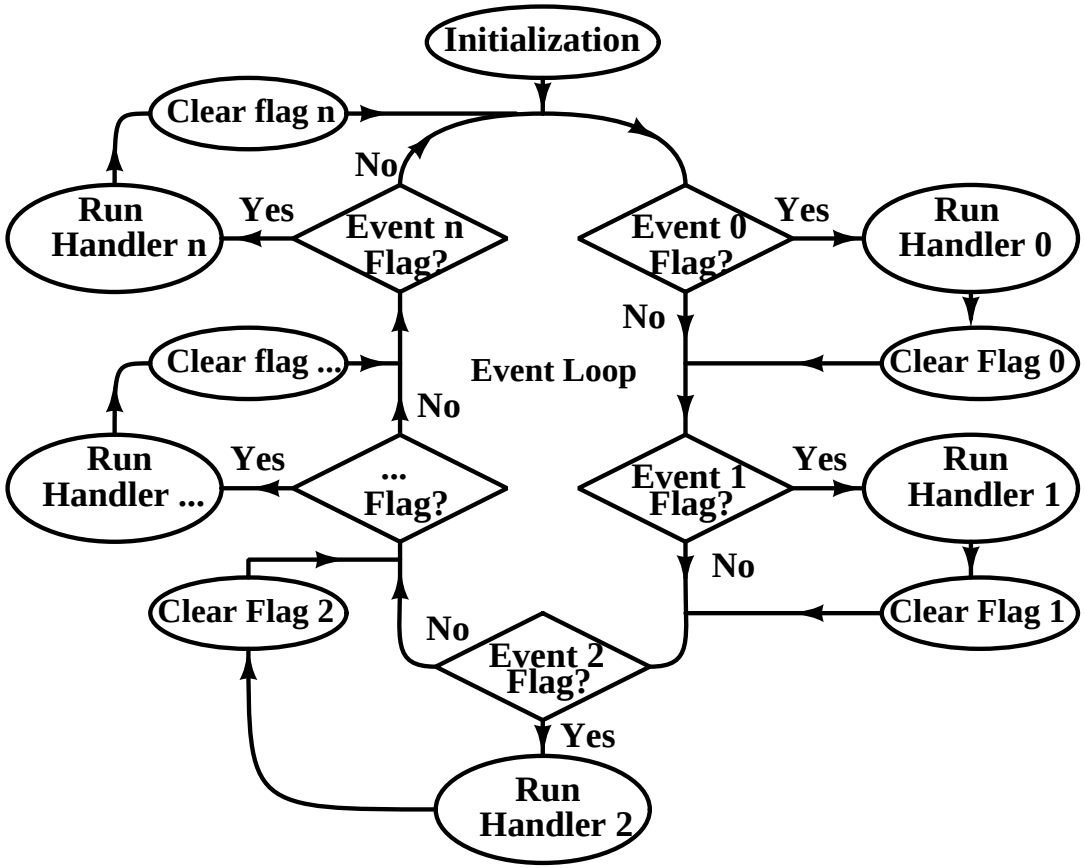


Each event is designed to set a “Flag” when it occurs.

(A flag is just a bit or a variable which is given a recognizable value when the event occurs).

The program runs in an endless loop, checking if the flag for any event is set.

Structure of an Event Driven Program



- If any event flag is set, it runs the corresponding handler, clears the flag and re-joins the loop.
- If the flag is not set, it just goes and examines the next event flag and so on . . . in an endless loop.

A program with such a structure is said to be **event driven**.

Event Driven Programs with Arduino

- The software structure in the Arduino Integrated Development Environment (IDE) is optimized for event driven programs.
- The in-built main program (in C or C++) first calls a user supplied function called **setup( )**.
- All initialization code is to be put in this function by the user.
- The library included in the IDE provides many functions which make the user's task easy.

  For example, to make the digital pin 13 an output pin, one can just call the library function **pinMode** as:

  pinMode(13, OUTPUT);

  And to set up the serial communication to use 9600 baud as the data rate, one can call:

  Serial.begin(9600)

Digital and Analog I-O

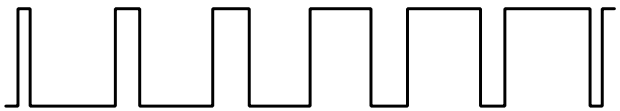
- After the user supplied function “**setup**” returns, the main program enters an endless loop.
- In each iteration of the loop, it calls another user supplied function called **loop( )**.
- The user can place all operational details relevant to continuous running of the application in this function.
- The library included in the IDE provides many functions which make it easy to input and output digital values.
- The micro-controller used in Arduino (AVR 328P) has an in-built 10 bit ADC with 6 input channels which can be used to input analog voltages. Library functions included in the IDE enable one to convert the analog input at any of the 6 channels to a 10 bit digital word.

Digital and Analog I-O

- The function **analogRead** provides the capability of reading the analog voltage on any of the analog input pins and converting it to a 10 bit digital value.
- **analogRead** accepts a channel number as its input and returns a 10 bit integer – which is the converted value from the ADC.
  For example,
  int sensorValue = analogRead(A0);
  declares the variable sensorValue to be a (16 bit) integer and places the 10 bit ADC value corresponding to the analog input on channel A0 in it.
- Channels A0 to A5 are available for ADC input.
- The library also has a mapping function which can map this 10 bit range onto a given range – say 0 to 100.

Digital and Analog I-O

AVR 328P does not have a built-in D to A converter. However, it can output a **Pulse Width Modulated (PWM)** digital stream on selected pins.



In Pulse Width Modulation, the ratio of durations of ON and OFF periods is varied.

- The ratio of ON time to total cycle time (ON duration + OFF duration) is called **Duty Cycle**.
- If this output is averaged, it will produce an analog value proportional to the duty cycle.
- Arduino IDE library provides functions for PWM output with the desired duty cycle.
- Explicit averaging may not be required for relatively slow devices such as motors. Also, human responses are slow enough so that the intensity of an LED driven by a PWM waveform will appear to be proportional to the duty cycle.

Digital and Analog I-O

Library function **analogWrite** permits writing PWM output to selected pins.

The function is somewhat inaccurately named – since the output is not really analog but a Pulse Width Modulated digital stream.

- The function takes the pin number and the duty cycle as its arguments. The duty cycle is provided as an integer between 0 and 255. (The actual duty cycle is the provided number/255).
- For example, **analogWrite(3, 128)** will output a PWM waveform on pin 3 with approximately equal ON and OFF times.
- A low pass filter can be connected to the pin to provide explicit averaging if required.
- Of course an external D to A converter can always be used if more accurate analog outputs are required.

Manifest Constants defined in Arduino Library

The Arduino library defines several symbolic names for constants. These are known as **manifest** constants (because they make the function of these constants clear).

Frequently used manifest constants are:

HIGH | **LOW :** represent the digital state of a pin.

INPUT | **OUTPUT** | **INPUT_PULLUP:** represent the mode in which a pin is to be used.
(**INPUT_PULLUP** is an input for which an internal pull up is enabled. This permits driving it with open collector outputs.)

LED_BUILTIN: is the pin number to which the on-board LED is connected. For Uno and Nano boards, this is pin number 13.

true | **false:** represent the logic value.

Library functions in Arduino Library: Digital I-O

The Arduino library provides many functions which permit one to use the functionality provided by the micro-controller.
We have seen many of these already.

Digital I/O: **digitalRead(pin);** returns a **HIGH** or **LOW** value depending on the voltage level on the pin.

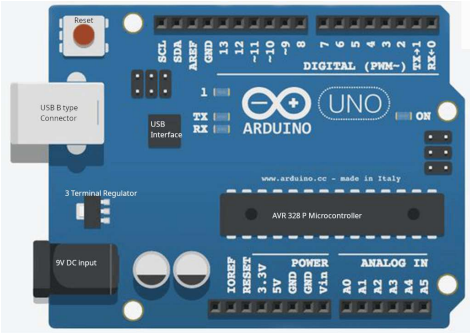
digitalWrite(pin, value); writes **HIGH** or **LOW** to a pin.

pinMode(type); sets the direction for I-O on a digital pin. The type can be **INPUT**, **INPUT_PULLUP** or **OUTPUT**.

If the type is **INPUT_PULLUP**, an internal pull up circuit is attached to the input pin. This is useful when the external driver is of open collector or open drain type.

Arduino Library Functions for Analog I-O

int analogRead(pin); returns the 10 bit ADC value corresponding to the analog voltage on the given pin.
void analogWrite(pin, PWMvalue); will output a PWM waveform on the given pin with the specified duty cycle (range 0 to 255).
void analogReference (type); configures the reference voltage used for A to D conversion. The argument “type” can be:



DEFAULT – to use the default analog reference of 5 volts (on 5V Arduino boards) or
INTERNAL – to use a built-in reference equal to 1.1 volts (= silicon band gap), or
EXTERNAL – to use the voltage applied to the AREF pin (0 to 5V only) as the reference.

This function must be called ***before*** analogRead to avoid shorting internal and external references.

Arduino Library Functions for Timing and Delay

Timing: **void delay(d_milli);** delays the execution of the following statements of the program by the given number of milliseconds.

void delayMicroseconds(d_micro) is similar to delay, but the argument is interpreted as microseconds.

int millis( ); returns the number of milliseconds which have elapsed since the start of the program.

int micros( ); returns the number of microseconds which have elapsed since the start of the program.

Some of these functions use the internal timers and interrupts for their operation. Others use software timing. Use of interrupts in the user program can interfere with their functioning.

Library functions in Arduino Library

Other I/O: **void tone(pin, frequency, [duration]);** A call to this function generates a square wave on the given pin with the specified frequency for the given duration. The third argument is optional. If duration is not given, the output will continue till a call is made to the function:
void noTone(pin);

Only one frequency can be generated at a time. Multiple frequencies cannot be generated on different pins using this function.

Minimum frequency which can be generated is 31 Hz. (This limitation comes from the maximum divider value which can be loaded in the timer/counter chip).

This function should not be used along with PWM output, because both use the same resources.

Interrupting the processor in Arduino

Interrupts in Arduino An interrupt stops the main program in order to run a specified function. The main program resumes from where it was stopped when the specified function returns.

A few library functions are used for managing interrupts. **void noInterrupts( )** disables interrupts from occurring, while **void interrupts( )** enables interrupts.

On Uno and Nano cards, external devices can interrupt the running program by sending a pulse on digital pin 2 or on digital pin 3.

Internal peripherals such as timers etc. also use interrupts.

Therefore care has to be taken while disabling interrupts – it may interfere with functions like serial communication.

Interrupting the processor in Arduino

Attaching an interrupt Function **void attachInterrupt( )** is used for causing an interrupt when a signal of a specified type is seen on digital pins 2 or 3 in an UNO or Nano card.

The function takes 3 arguments. The first argument is the interrupt number, the second is the name of the function to be run when an interrupt occurs and the third argument specifies the kind of signal on the interrupt pin which will result in an interrupt.

The interrupt number should not be given directly, but should be determined from a call to the function **digitalPinToInterrupt(pin)**.

The returned value from this function should be given as the first argument to **attachInterrupt**.

Interrupting the processor in Arduino: function arguments

Attaching an interrupt: contd. The second argument is the name of the user supplied function which should be run after stopping the main program.

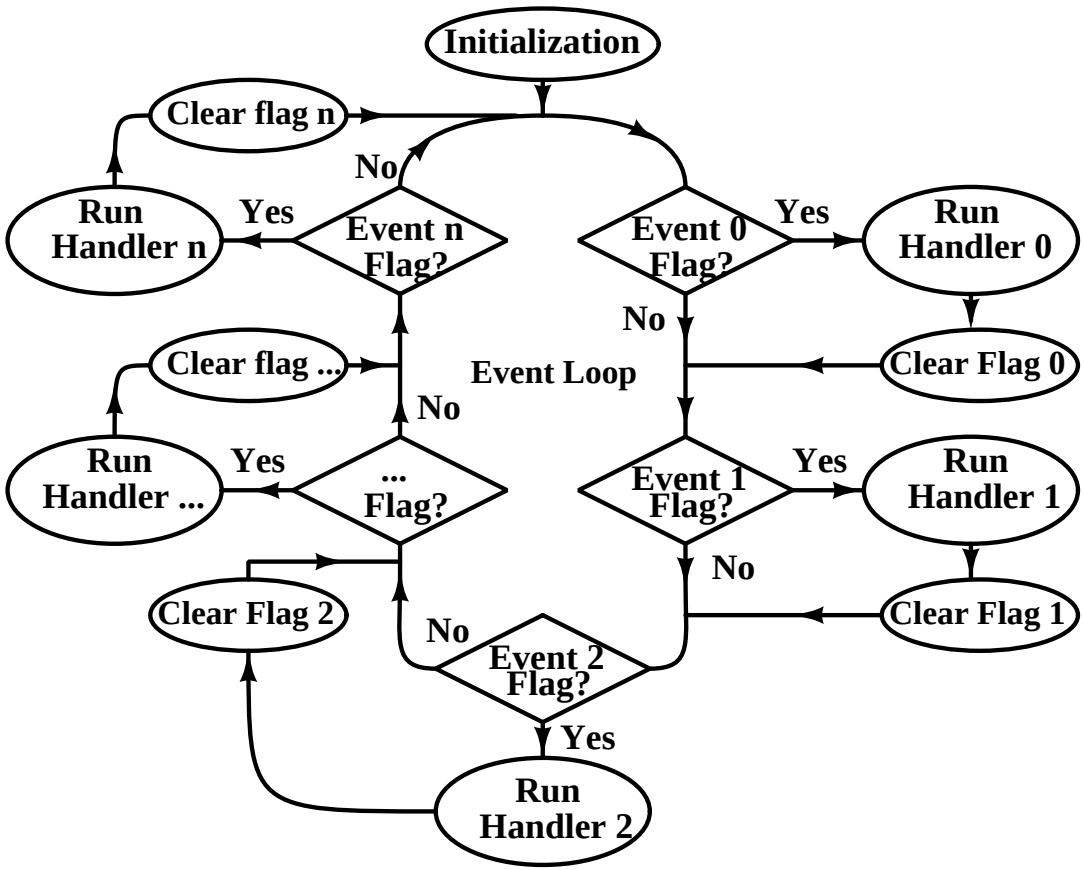
The last argument specifies the type of input on pin 2 or 3 which will result in an interrupt. This argument can be one of pre-defined manifest constants **LOW, CHANGE, RISING** or **FALLING**.

LOW interrupts when the signal is '0', **RISING** interrupts when the signal changes from '0' to '1', **FALLING** interrupts on a '1' to '0' transition and **CHANGE** interrupts when either of these transitions occurs.

Once the interrupt is recognized, further interrupts are disabled till the specified function has run.

Interrupting the processor in Arduino: Handler function

The function run after interrupting the main program should be short and the processor should remain in interrupted state for as small a time as possible.

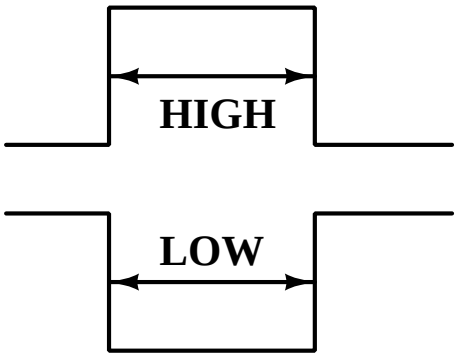


Rather than running a detailed computation as the interrupt function, one uses the interrupt function to just set an event flag.

The handling of the event then occurs in the main program in the event loop as outlined in the beginning of this lecture.

Pulse width Measurement

Pulse width measurement



pulseIn( );, **pulseInLong( );** – these functions are used for measuring the width of a high or a low pulse. These take two arguments - the pin number where the pulse will be applied and whether the HIGH of LOW duration is to be measured.

The function **pulseIn** is suitable for use with interrupts disabled. (The software timing used by this function stops during interrupts and so the returned value will be inaccurate if interrupts occur).

The function **pulseInLong( )** uses hardware timing with interrupts. So the interrupts must be enabled when this function is called.

Pulse width Measurement

pulseIn( ) is more useful for short pulses since we can afford to disallow interrupts for a short time.

As its name implies, **pulseInLong( )** is more suitable for long pulses. However, it may interfere with serial communications and delay functions.

The syntax for calling these functions is:
pulseIn(pin, HIGH); or **pulseIn(pin, LOW);** and **pulseInLong(pin, HIGH);** or **pulseInLong(pin, LOW);**.

pin is the pin number on which the pulse will arrive. The second argument decides whether the width of the **HIGH** part of the pulse will be measured or the **LOW** part.
These functions have an optional third argument which specifies the maximum time for which the function will wait for the pulse to end.
If not provided, this time is taken to be 1 second.

Other Libraries

- We have taken a brief tour of many of the functions provided by the built-in library of Arduino IDE.
- There are hardware cards which plug directly into the connectors of Arduino Uno. These are designed for specific functions – such as driving stepper motors, speed control for battery operated (BO) motors, communicating using bluetooth etc.
  These cards are called **shields** (because of the way they sit on the Arduino card).
- Libraries to support these shields are available. One can download these libraries and include them in the IDE.
- Since these libraries are mostly developed by hobbyists and placed in the public domain, documentation may be vague or non-existent.
  However, the source code for functions included in these libraries is available and one can figure out how to use these functions for projects.

Application Development with Arduino

Application development using Arduino boards involves:

1. Choice of appropriate additional hardware,
2. Development of required algorithms
3. Implementation of algorithms in software

Many of the hardware techniques such as negative feedback, hysteresis using Schmitt triggers etc. have their counterparts in software.

We'll review some of these briefly.

Application Development with Arduino

To illustrate some of the algorithms used for application development, we'll use temperature control of a water bath as an example.

We would like to keep the temperature of water in the bath as close as possible to a given temperature (called **set point**) which is higher than the room temperature (no cooling required) and which should be programmable.
Let us first look at the hardware required for this:

- We obviously need a heater. To control the temperature, we'll require some means of controlling power to the heater. This can be done through
  1. ON/OFF control through a relay, or
  2. Pulse Width control through thyristors driven from Arduino, or
  3. Voltage control through a programmable power source to the heater.
- Accordingly, we shall need drivers for relays/thyristors/DC source which will be driven using digital signals from Arduino.

Application Development: ON/OFF control

We also require hardware for measuring the actual temperature (using LM35 temperature sensor or a thermocouple), and means for adjusting the set point (through a potentiometer or dialing it in through a keyboard).

Let us first consider the simplest option – that of ON/OFF control.

- We can measure the temperature at regular intervals and compare it with the set point. If the actual temperature is higher than the set point, we turn the relay OFF, if it is lower, we turn it ON.
- When the temperature is close to the set point, a small amount of heating takes it above the set point which turns the relay OFF. However, then the temperature quickly drops below the set point, which turns the relay ON. This causes the relay to “chatter”.

How to stop the relay from chattering due to this frequent switching?

Application Development: ON/OFF control

To prevent the relay from chattering, we can use a small amount of hysteresis in the decision to turn the relay ON or OFF.

- Instead of comparing the temperature with a single set point, we use a “high limit” and a “low limit” on either side of the set point.
- We turn the relay ON only if the temperature is below the “low limit”. We turn if OFF only if the temperature is above the “high limit”.
- We now have a trade off – if the high and low limit are too close to the set point, the relay will turn ON and OFF fequently.
- If these limits are set far from the set point, the temperature will ramp between these two limits and the worst case error between the actual temperature and the set point will be high.

You would have noticed this kind of control in many electric irons.

Application Development: Proportional Control

A smoother way to control the temperature of the bath would be to apply power to the heater proportional to the error in temperature –

- The farther away we are below the set point, higher is the power applied.
- However as we approach the set point, the error becomes less and lower power will be applied.
- If we are at or above the set point, no power will be applied.

This results in more accurate control of temperature.

However, the power applied when we reach the set point is zero!
So the bath will always start cooling down due to heat losses as soon as it reaches the set point.

Proportionate control will always settle at non-zero error!
(A work around is to calculate the error from a point just above the set point, so that the power is non zero at the set point).

Application Development: Integral Control

We don't want to reduce the power to zero when we reach the set point. We want the heater to apply *constant* power, which keeps the bath at the set point.

- What operation gives a constant result when its argument reaches zero? The integral, of course!
- What we should do is to integrate the error and apply power proportional to this integral. Now when the error becomes zero, a constant power will be maintained which will keep the bath at the set temperature.
- If we overshoot the set point, error will become negative and the integral will reduce in value. With lower applied power, we'll come back to the set point and keep the power at this lower value.
- Since the proportional term is zero at the set point any way, we can use a value for applied power which is the weighted sum of proportional term and integral term. This is known as Proportional-Integral or P-I control.

Application Development: PI Control

The integration of error should be taken over how much time?

- Temperature error which was there long ago may be less relevant for determining the power to be applied at the current instant.
- Therefore we integrate the error over the recent history. Contribution due to error from long ago needs to be dropped out from the integral. This is known as the **reset rate**.
- Proportionality constants for the P and I contributions, as well as the reset rate have to be tuned for the specific system being controlled.

PI control works reasonably well for keeping the bath temperature constant. However, it is slow to react to sudden changes in the error. To take care of this, we add a third term – the differential term.

Application Development: PID Control

- PI control is very slow to react to sudden changes in error.
- For example, if we change the set point, suddenly there is a large error from the current temperature. PI control will be sluggish to react to it.
- To take care of sudden changes in the error, we need a term which will be proportional to the *rate of change* of the error. This can be provided by a term proportional to the differential of the error.
- When we include the differential term, the control strategy is called Proportional-Integral-Differential – or PID control.

PID control is widely used for controlling various process parameters.

How do we implement it in a micro-controller based system?

Implementing PID Control

- We measure the temperature at regular intervals, compute the error and store these values in an array.
- Every time we compute the current value of error, we add it to a moving sum and subtract the error which had been added “n” steps before. (“n” is the reset rate).
- We also compute the difference between the current and previous errors and use it as the differential term.
- We compute a weighted sum of the current error, current value of the moving sum and the differential term by multiplying each of these with their proportionality constants and adding them.
- We output this weighted sum to the external hardware which will apply power proportional to this value.

This algorithm is not restricted to temperature control. It can be applied to any parameter which has to be kept at a set value.
For example, it could be used for a line follower robot where we control the orientation to keep the robot centered on the line.

That is all folks!