1 Review of Digital Electronics

1.1 Gate Types and Truth Tables

The basic logic gates are AND, OR, NAND, NOR, XOR, INV, and BUF. The last two are not standard terms; they stand for “inverter” and “buffer”, respectively.

Truth Tables are an easy way to represent a combinational logic output by tabulating all possible inputs. The symbols for these gates and their corresponding Boolean expressions and truth tables are given below.

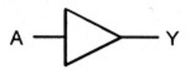
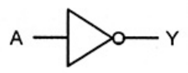
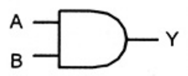
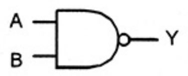
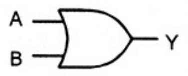
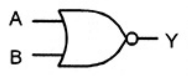
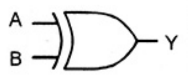
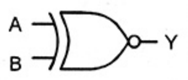
Logic function	Logic symbol	Truth table	Boolean expression
Buffer		A Y 0 0 1 1	$Y = A$
Inverter (NOT gate)		A Y 0 1 1 0	$Y = \overline{A}$
2-input AND gate		A B Y 0 0 0 0 1 0 1 0 0 1 1 1	$Y = A \bullet B$
2-input NAND gate		A B Y 0 0 1 0 1 1 1 0 1 1 1 0	$Y = \overline{A \bullet B}$
2-input OR gate		A B Y 0 0 0 0 1 1 1 0 1 1 1 1	$Y = A + B$
2-input NOR gate		A B Y 0 0 1 0 1 0 1 0 0 1 1 0	$Y = \overline{A + B}$
2-input EX-OR gate		A B Y 0 0 0 0 1 1 1 0 1 1 1 0	$Y = A \oplus B$
2-input EX-NOR gate		A B Y 0 0 1 0 1 0 1 0 0 1 1 1	$Y = \overline{A \oplus B}$

Figure 1: Basic Logic gates

1.2 Boolean Algebra and K-Maps

Boolean algebra can be used to formalize the combinations of binary logic states. Using the definition of Boolean addition, multiplication and inversion, we can define all the logic gates algebraically.

For designing any combinational circuit, we use Boolean algebra. However, for any arbitrary circuit the boolean expression might be lengthy and cumbersome which might lead to inefficient implementation. Thus, the need of logic minimization. One method is through the use of Karnaugh Maps or K-Maps.

For a boolean function of n variables, $x_1, x_2, \ldots x_n$, a product term in which each of the n variables appears once (in either its complemented or uncomplemented form) is called a *minterm*. The addition or "OR"-ing of minterms give the *Sum of Products*.

For a boolean function of n variables, $x_1, x_2, \ldots x_n$, a sum term in which each of the n variables appears once (in either its complemented or uncomplemented form) is called a *maxterm*. The multiplication or "AND"-ing of maxterms give the *Product of Sums*.

1.3 Multiplexer

A multiplexer (MUX) is a device which passes one of several data inputs to one output. Generally there are 2^n data inputs and n control lines which determine which input is steered to the output.

Hence, a MUX can take many data bits and put them, one at a time, on a single output data line in a particular sequence. This is an example of transforming parallel data to serial data.

By adding gate-level circuitry to MUX inputs, any arbitrary combinational function can be realised with a 2:1 MUX. Also, any n variable combinational function can be implemented with a $2^n : 1$ MUX, $2^{n-1} : 1$ MUX and so on.

1.4 Decoder

Decoder (DEC) is basically, a combinational type logic circuit that converts the binary code data at its input into an equivalent decimal code at its output. Generally there are n inputs and 2^n outputs. Depending on the input, the decoder activates only one of the

2^n outputs.

Therefore, whichever output line is “HIGH” identifies the binary code present at the input, in other words it “de-codes” the binary input and these types of binary decoders are commonly used as Address Decoders in microprocessor memory applications.

1.5 Priority Encoder

An encoder is a combinational logic circuit that accepts an active level on one of its inputs (inputs represents digits, such as decimal, octal and so on) and converts it to a coded output. An encoder has 2^n input lines, only one of which is activated at a given time and produces an n-bit output code, depending on which input is activated.

However, if more than one input are active simultaneously, the output is unpredictable. This ambiguity is resolved if priority is established so that only one input is encoded, no matter how many inputs are active at a given point of time.

The priority encoder includes a priority function. The operation of the priority encoder is such that if two or more inputs are active at the same time, the input having the highest priority will take precedence.

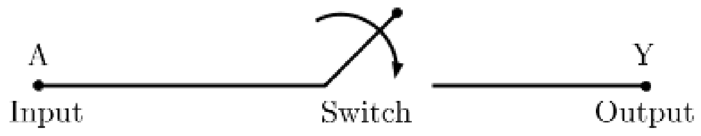
1.6 Latches and Flip-Flops

Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal. After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.

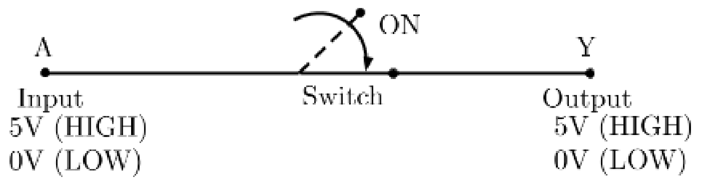
2 Tristate concept and Bus structure

2.1 Tristate logic

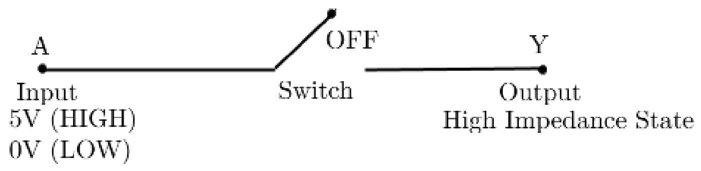
In digital logic, we have two states: HIGH and LOW. Let us consider the switch connected between input (A) and output (Y) as shown below.



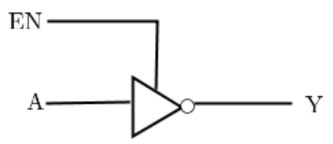
Now let us switch it ON and apply 5V (HIGH) to the input. Since the switch is ON, input goes to the output and we get 5V as output. Similarly, when we apply 0V (LOW) as the input, we get 0V as output.



However, when the switch in OFF, we cannot say that the output is at 0V (LOW). This new state, when the input is not connected to the output, is called a High-Impedance State (Z).



Thus, any logic device which has the provision to effectively remove its influence from the circuit (usually through an ENABLE input) exhibits a tristate logic. For example, let us take a NOT gate as shown. The gate works as usual when the EN input is HIGH, but when EN is set to LOW, irrespective of the input, the output is at High-impedance state.



EN	A	Y
1	0	1
1	1	0
0	X	Z

2.2 Understanding Bus

A group of signals or wires is called a Bus. The purpose of a bus structure together with the tristate logic is to reduce the number of connections needed for transfer of data between the components. A bus is usually represented as shown below.

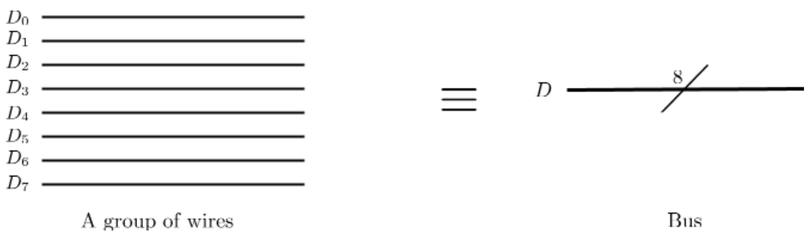


Figure 2: Representation of a bus

When building a digital system, the number of interconnections between different components becomes a significant problem. Let us take an example of a system of four 8-bit components **A**, **B**, **C** and **D** where data transfer between any two components is permissible. A total of 12 "data highways" or bus as shown below are needed, each containing 8 wires.

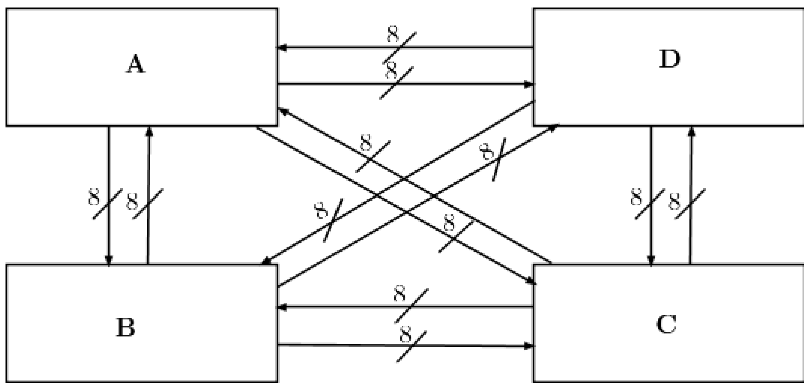


Figure 3: Interconnection of 4 8-bit components

As more components are added to the system, the number of interconnections increases and the design would become very complex. This interconnection problem can be dramatically reduced by using tristate logic and a common bus structure.

Instead of having interconnections between each of the components separately, each component can be connected to a common bus. Of course this can only work if each device knows when to "speak" (give some data) and when to "listen" (take some data). Otherwise the devices could try to access the common bus at the same time leading to garbage data (*bus contention*). Thus, each component needs to be designed with a tristate logic, i.e., only when it is enabled, the component can give or take data and when it is disabled it is at high impedance state (disconnected from the common bus).

The device that determines who will speak and who will listen is called the *bus master*. Usually, the Control Unit of the processor does this job. Depending on our instruction, the control unit will generate a control signal whereby only the required components are enabled for data transfer.

For example, let us consider a data transfer from component **C** to **A**. The control unit will first activate **C** and allow data from **C** to be outputted to the common bus. Now this data is available at the inputs of all the components. But according to our instruction, the control unit will activate only the component **A** and allow it to take the data from the common bus.

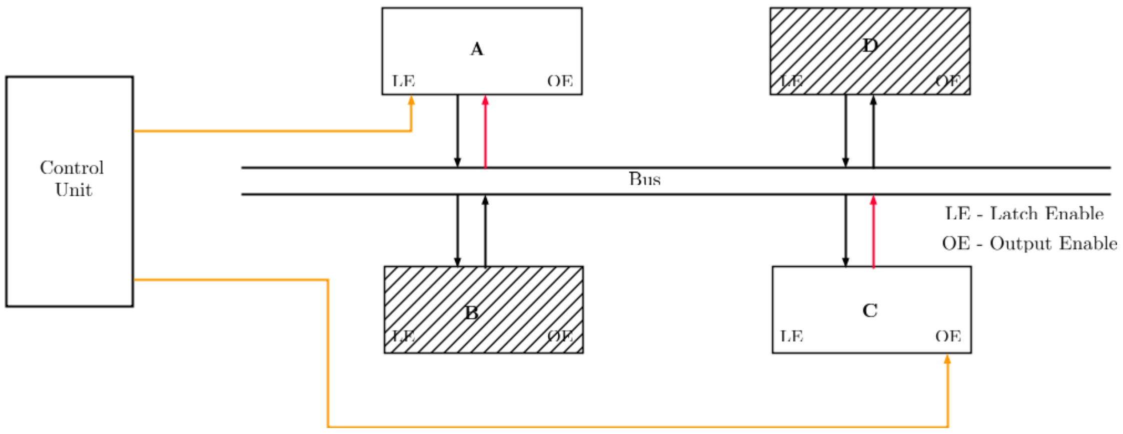


Figure 4: Data transfer using a bus structure

3 Microprocessors

3.1 Basic Microprocessor

A microprocessor is a

- multipurpose – addition, subtraction, comparison, etc.,
- programmable – depending on user's preference, it can be instructed to perform given tasks,
- clock-driven – synchronous circuit,
- register-based – to store the data temporarily for processing

electronic device that reads binary instructions from a storage device called *memory*, accepts data as input and processes data according to those instructions, and provides results as output.

A typical microprocessor can be represented in block diagram as shown in figure below.

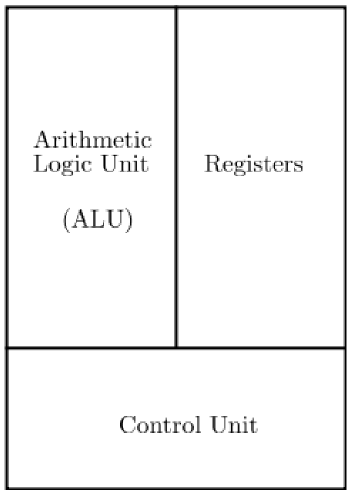


Figure 5: Block diagram of a basic microprocessor

All the processing of a microprocessor can be boiled down to either arithmetic or logical functions. These functions are performed by the **ALU**.

Any arithmetic or logical operation needs operands. The **Registers** are used primarily to store, temporarily, the data required for the execution of various operations.

The **Control Unit** provides the necessary timing and control signals to all the operations in the microprocessor. It basically tells when to do what.

3.2 Basic Microprocessor based System

A microprocessor on its own cannot perform its various tasks. It needs some device to give it instructions (input), some device to show the results of the instructions (output) and some device to store the instructions (memory).

So, any microprocessor based system can be represented in block diagram as shown in figure below.

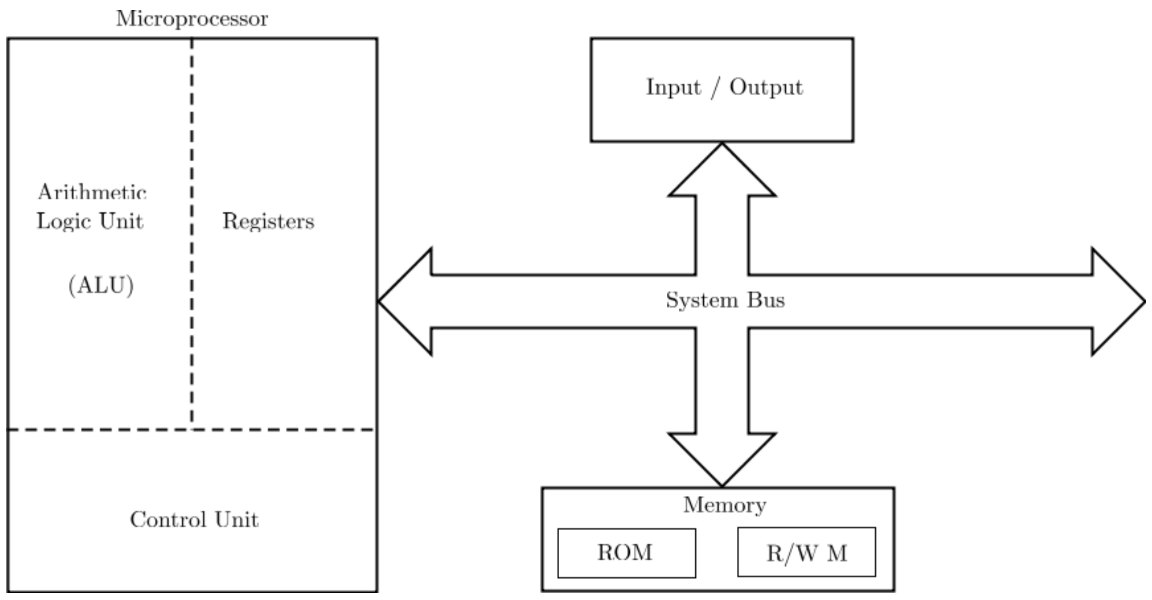


Figure 6: Block diagram of a basic microprocessor based system with bus architecture

It includes three components: *microprocessor*, *I/O device* and *memory* (read/write memory and read-only memory). These components are connected to each other via *System Bus*.

3.3 Basic working of a microprocessor

The sequence of process a microprocessor does when it needs to perform a task can be summarised as: *fetch*, *decode* and *execute*.

Let us assume some instructions for a task are stored in a memory. When we command the microprocessor to perform the task, it “goes” to memory to get the first instruction (fetch), understands the instruction (decode) and then performs according to the instruction (execute).

This sequence of *fetch*, *decode* and *execute* is continued until the microprocessor is instructed to *stop* or *halt*.

During the entire process, the microprocessor uses the system bus to fetch the instructions and data from the memory. It uses registers to store data temporarily and it performs the computing function in the ALU.

3.4 Applications of Microprocessor and its versatility

Applications of microprocessor are discussed in the next section.

Versatility of microprocessor comes from its programmability feature. Microprocessors are designed as *stored-program* computers. This means that by keeping the same hardware, we can load or store different types of programs/softwares and thus use the same microprocessor for numerous applications.

3.5 Evolution of microprocessors

The evolution of microprocessor can be traced along three branches:

- General Purpose Processor Branch: These are typically used for making computers. Applications include personal computers (PC), laptops, workstations, servers, etc.
- Microcontroller Branch: These are typically used for making embedded systems. Applications include digital cameras, microwave ovens, washing machines, digital oscilloscope, etc.
- Special Purpose Processor Branch: These are usually used for some specific purpose like Digital Signal Processing, Communication, etc. Applications include Switches, Routers, etc.

As the microprocessors were evolving, it moved from 4-bit to 8-bit and so on and the current generation is 64-bit. Along with that, the processing power and features were also increased.

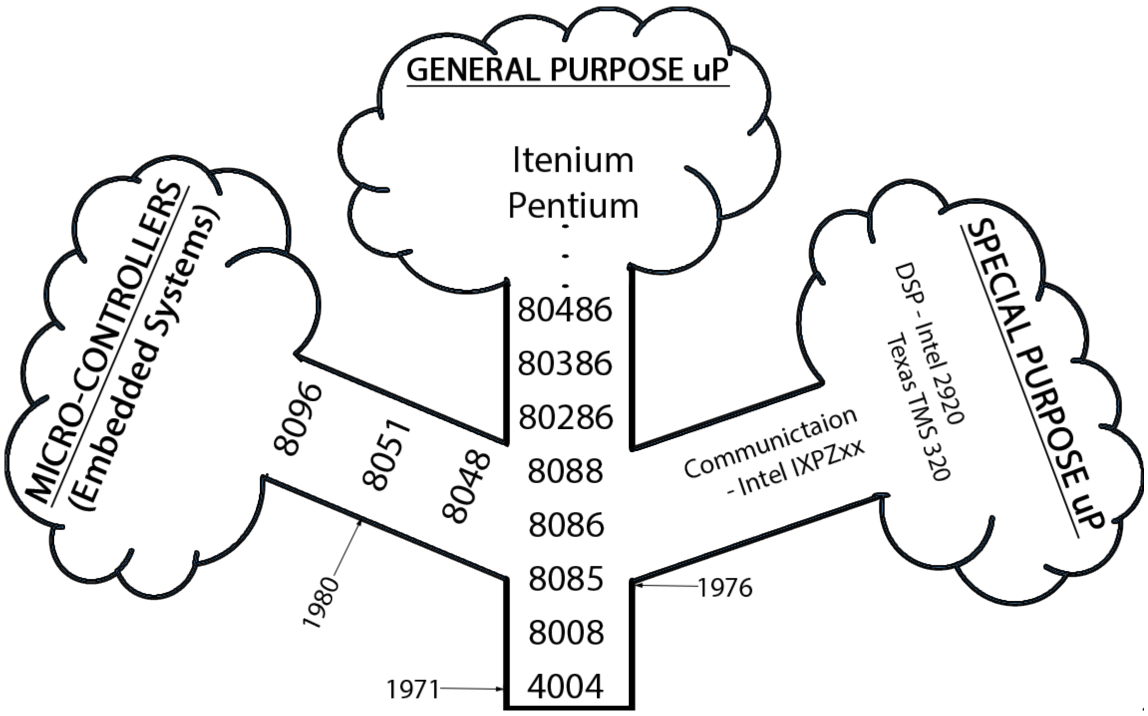


Figure 7: Microprocessor evolution tree (Taken from NPTEL Lectures)