When you write `add $t0, $t1, $t2`, you imagine something like this:

What kind of hardware can ADD two binary integers?

We need to understand about GATES and BOOLEAN ALGEBRA, that are foundations of logic design.
**AND gate**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X.Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR gate**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X+Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOT gate**

<table>
<thead>
<tr>
<th>X</th>
<th>(\bar{X})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Typically, logical 1 = +3.5 volt, and logical 0 = 0 volt. Other representations are possible.
Analysis of logical circuits

What is the value of F when X=0 and Y=1?

Draw a truth table.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This is the exclusive or (XOR) function. In algebraic form \( F = \bar{X}.Y + X.\bar{Y} \)
More practice

1. Let $\overline{A}.B + A.C = 0$. What are the values of $A, B, C$?

2. Let $(A + B + C) \cdot (\overline{A} + \overline{B} + C) = 0$. What are the possible values of $A, B, C$?

- Draw truth tables.
- Draw the logic circuits for the above two functions.
Elementary Boolean Algebra

\[
\begin{align*}
A + \bar{A} &= 1 \\
A \cdot \bar{A} &= 0
\end{align*}
\]

\[
\begin{align*}
1 + A &= 1 \\
1 \cdot A &= A
\end{align*}
\]

\[
\begin{align*}
0 \cdot A &= 0 \\
0 + A &= A
\end{align*}
\]

\[
\begin{align*}
A + A &= A \\
A \cdot A &= A
\end{align*}
\]

\[
\begin{align*}
A \cdot (B + C) &= A \cdot B + A \cdot C & \text{Distributive Law} \\
A + B \cdot C &= (A+B) \cdot (A+C)
\end{align*}
\]

\[
\begin{align*}
A \cdot B &= \bar{A} + \bar{B} & \text{De Morgan's theorem} \\
A + B &= A \cdot B
\end{align*}
\]
Synthesis of logic circuits

Many problems of logic design can be specified using a truth table. Give such a table, can you design the logic circuit?

Design a logic circuit with three inputs A, B, C and one output F such that F=1 only when a majority of the inputs is equal to 1.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum of product form

\[ F = \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C \]

Draw a logic circuit to generate F
Simplification of Boolean functions

Using the theorems of Boolean Algebra, the algebraic forms of functions can often be simplified, which leads to simpler (and cheaper) implementations.

Example 1

How many gates do you save from this simplification?
Example 2

\[ F = \overline{A.B.C} + A.B\overline{C} + A.B.C + A.B.C \]

\[ = \overline{A.B.C} + A.B\overline{C} + A.B.C + A.B.C + A.B.C + A.B.C \]

\[ = (A.B.C + A.B.C) + (A.B.C + A.B.C) + (A.B.C + A.B.C) \]

\[ = (A + A).B.C + (B + B).C.A + (C + C).A.B \]

\[ = B.C + C.A + A.B \]

Example 3  Show that \( A + A.B = A \)

\[ A + AB \]

\[ = A.1 + A.B \]

\[ = A. (1 + B) \]

\[ = A.1 \]

\[ = A \]
Simplification using Karnaugh Maps

Follow the class lectures to understand how to simplify Boolean functions using K-maps. Several examples will be worked out in the class.
Other types of gates

NAND gate

\[ A \cdot B \]

NOR gate

\[ A + B \]

Be familiar with the truth tables of these gates.

Exclusive OR (XOR) gate.

\[ A \oplus B = A \cdot B + A \cdot B \]
NAND and NOR are universal gates

Any function can be implemented using only NAND or only NOR gates. How can we prove this?

(Proof for NAND gates) Any boolean function can be implemented using AND, OR and NOT gates. So if AND, OR and NOT gates can be implemented using NAND gates only, then we prove our point.

1. Implement NOT using NAND
2. Implementation of AND using NAND

3. Implementation of OR using NAND

(Exercise) Prove that NOR is a universal gate.
**Example (to be worked out in class)**

How to convert any circuit that uses AND, OR and NOT gates to a version that uses NAND (or NOR gates only)?

**Additional properties of XOR**

XOR is also called modulo-2 addition. Why?

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$A \oplus B = 1$ only when there are an odd number of 1’s in $(A,B)$. The same is true for $A \oplus B \oplus C$ also.

$1 \oplus A = \overline{A}$

Why?

$0 \oplus A = A$
Logic Design Exercise

Half Adder

\[ S = A \oplus B \]
\[ C = A \cdot B \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{Half Adder} \]
**Full Adder**

![Full Adder Diagram]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S = A \oplus B \oplus C_{in} \]

\[ C_{out} = A.B + B.C_{in} + A.C_{in} \]

How can you add two 32-bit numbers?
**Combinational vs. Sequential Circuits**

**Combinational circuits.**

The output depends only on the current values of the inputs and not on the past values. Examples are adders, subtractors, and all the circuits that we have studied so far.

**Sequential circuits.**

The output depends not only on the current values of the inputs, but also on their past values. These hold the secret of how to memorize information. We will study sequential circuits later.
Decoders

A typical decoder has n inputs and $2^n$ outputs.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A 2-to-4 decoder and its truth table.

Draw the circuit of this decoder.

The decoder works per specs when (Enable = 1). When Enable = 0, all the outputs are 0.

Exercise. Design a 3-to-8 decoder.


**Encoders**

A typical encoder has $2^n$ inputs and $n$ outputs.

A 4-to-2 encoder and its truth table.

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplexor

It is a many-to-one switch, also called a selector.

A 2-to-1 mux

\[ F = \overline{S} \cdot A + S \cdot B \]

Exercise. Design a 4-to-1 mux.
Another design of a decoder

Exercise 1. Design a 2-to-4 decoder using 1-to-2 decoders only.

Exercise 2. Design a 4-to-1 multiplexor using 2-1 multiplexors only.

To be discussed in the class.
Demultiplexors

A demux is a one-to-many switch.


So, \( X = \overline{S} \cdot A \), and \( Y = S \cdot B \)

Exercise. Design a 1-4 demux.

We will discuss the design of a 1-bit ALU in class.
A 1-bit ALU

Understand how this circuit works.

Then look at the diagram of the 32-bit ALU in p.235.

Need to add one more input to the mux to implement **slt**
Converting an adder into a subtractor

\[
A - B \quad \text{(here - means arithmetic subtraction)}
= A + \text{2’s complement of } B
= A + \text{1’s complement of } B + 1
\]

1-bit adder/subtractor

For subtraction, B invert = 1 and Carry in = 1
1-bit ALU for MIPS

Assume that it has the instructions add, sub, and, or, slt.

Less will be used to detect if the 32-bit number $A$ is less than the 32-bit number $B$. See the next page.

If $A < B$ then Set = 1 else Set = 0
A 32-bit ALU for MIPS

B invert       C in operation

A0              ALU
B0
Less

A1              ALU
B1
O

A31              ALU
B31
O

Result 31
Set
overflow
**Fast Carry Propagation**

During addition, the carry can trigger a “ripple“ from the LSB to the MSB. This slows down the speed of addition.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

How to overcome this? Consider the following:

\[
c_1 = a_0.b_0 + a_0.c_0 + b_0.c_0
\]
\[
= a_0.b_0 + (a_0 + b_0).c_0
\]
\[
= g_0 + p_0.c_0 \text{ (where } g_0 = a_0.b_0, p_0 = a_0+b_0)\]

\[
c_2 = a_1.b_1 + (a_1 + b_1).c_1
\]
\[
= g_1 + p_1.(g_0 + p_0.c_0)
\]
\[
= g_1 + p_1.g_0 + p_1.p_0.c_0
\]

\[
c_4 = g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.c_0
\]
It will be complex. But you can use a two-level circuit to generate $c_4$. This will expedite addition. But it is impractical due to the complexity.

Practical circuits use a two-level approach. See the example of the 16-bit adder, designed from four 4-bit adders in p.246. Let

\[
\begin{align*}
G_0 &= g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 \\
G_1 &= g_7 + p_7.g_6 + p_7.p_6.g_5 + p_7.p_6.p_5.g_4 \\
G_2 &= g_{11} + p_{11}.g_{10} + p_{11}.p_{10}.g_9 + p_{11}.p_{10}.p_9.g_8 \\
G_3 &= g_{15} + p_{15}.g_{14} + p_{15}.p_{14}.g_{13} + p_{15}.p_{14}.p_{13}.g_{12} \\
\end{align*}
\]

\[
\begin{align*}
P_0 &= p_3.p_2.p_1.p_0 \\
P_1 &= p_7.p_6.p_5.p_4 \\
P_2 &= p_{11}.p_{10}.p_9.p_8 \\
P_3 &= p_{15}.p_{14}.p_{13}.p_{12} \\
\end{align*}
\]
Then

\[
\begin{align*}
C_1 &= G_0 + P_0.c_0 \\
C_2 &= G_1 + P_1.G_0 + P_1.P_0.c_0 \\
C_3 &= G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \\
\end{align*}
\]

This is implemented in the **carry look-ahead adder**.

See the figure in p. 246 of your textbook.

How much faster is the carry look-ahead adder?
Multiplication

Be familiar with shift operations first

1 0 0 1  (multiplicand)
0 1 0 1  (multiplier)

<table>
<thead>
<tr>
<th>0 1 1 1 0 1 0 0</th>
<th>1 0 1 0 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shift right</strong></td>
<td><strong>shift left</strong></td>
</tr>
</tbody>
</table>

0 0 1 1 1 0 1 0

0 1 0 0 0 1 1 1

See the diagrams in page 254 and 255 of your textbook.
**Floating point representation**

A scheme for representing a number **very small** to **very large**. It is widely used in the scientific world. Consider, the floating point number

<table>
<thead>
<tr>
<th>Exponent E</th>
<th>Significand F</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-</td>
<td>x x x x y</td>
</tr>
<tr>
<td></td>
<td>y y y y y y y y y y y</td>
</tr>
</tbody>
</table>

In **decimal** it means \((+/-) 1. \text{yyyyyyyyyyyy} \times 10^{xxx}\)

In **binary**, it means \((+/-) 1. \text{yyyyyyyyyyyy} \times 2^{xxx}\)

(The 1 is implied)

**IEEE 754 representation**

<table>
<thead>
<tr>
<th>s</th>
<th>xxxxxxxx</th>
<th>yyyyyyyyyyyyyyyyyyyyyyy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

Largest \(= 1.111... \times 2^{+127} \approx 2 \times 10^{+38}\)

Smallest \(= 1.000... \times 2^{-128} \approx 1 \times 10^{-38}\)

These can be positive and negative, depending on \(s\).
IEEE 754 double precision (64 bits)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent</th>
<th>significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

Largest = \(1.111\ldots \times 2^{1023}\)
Smallest = \(1.000\ldots \times 2^{-1024}\)

What do you mean by **overflow** and **underflow** in FP?

An overflow occurs when the number if too large to fit in the frame. An underflow occurs when the number is too small to fit in the given frame.
### Biased Representation

Exponent = 1111111  \( 2^{-1} \) \quad \text{awkward for sorting}

Exponent = 0000000  \( 2^0 \)

However, to facilitate sorting, IEEE 754 treats 00...0 as the most negative, and 1,11..1 as the most positive exponent. This amounts to using a **bias** of 127.

\[
\begin{array}{c}
00000000 \texttt{(-127)} & 11111111 \texttt{(+128)} \\
\text{bias +127} \\
\text{smallest} & \text{largest}
\end{array}
\]

So, value = \((1)^s \times (1\text{-significand}) \times 2^{(exponent - \text{bias})}\)

Practice simple conversions from fraction to FP.
Sequential Circuits

The output depends not only on the current inputs, but also on the past values of the inputs.

An SR Latch

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q̅</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0/1</td>
<td>1/0</td>
<td>Old state continues</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Set state</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Reset state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Illegal inputs</td>
</tr>
</tbody>
</table>
A clocked D-latch

Clock is the enabler. If \( C = 0 \), \( Q \) remains unchanged.
When \( C = 1 \), then \( Q \) acquires the value of \( D \). We will use it as a building block of sequential circuits.

The main complaint is the “transparency”. An edge triggered circuit (or a master-slave circuit) solves this problem (to be discussed in the class).
A Master-Slave D flip-flop

The output Q acquires the value of D, only when one complete pulse (i.e. 0 1 0) is applied to the clock input. The external output Q changes after the falling edge.

A clock pulse

rising edge

falling edge
Register

A 32-bit register is an array of 32 D-flip-flops.

Data input

In the class, we will discuss about shift registers and counters. Then we will work out several exercises related to shift register, counters, and register transfers.
Binary counter

A toggle flip-flop (T)

Observe how Q3 Q2 Q1 Q0 changes when pulses are applied to the clock input of the leftmost flip-flop.
State diagram of a counter

A register file

How does it work? See diagrams B.18, B.19, B.20
A shift register can be used

- To divide (or multiply) a number by $2^k$;
- To generate control signals for controlling a sequence of operations (to be discussed in the class)
- For serial-to-parallel and parallel-to-serial data conversion

When the output of the rightmost FF is connected to the input of the leftmost FF, it becomes a circulating register. A circulating register that contains exactly a single 1 is called a ring counter.
Main Memory

Processor

MEMORY

Address

Data in

Data out

write

read

Read pp. B.26-B.35.

Memory

ROM (Read Only Memory)

RAM (Random Access Memory)

SRAM

DRAM
Typical sizes of SRAM are

\[(32 \text{ or } 64 \text{ or } 128 \text{ or } 256) \times (1 \text{ or } 2 \text{ or } 4 \text{ or } 8 \text{ bits})\]

How many lines are there in address, data in and data out?

Learn about bit lines and tri-state drivers (Fig. B.21, B.22).

Learn about how to design a large memory using smaller memory modules (Fig. B.23 and B.24). We will discuss these in the class.
32K x 1 bit RAM using 4 8K x 1 RAMs

For each chip, the write enable line is set to 1 during a write operation, and the output enable lines are set to 1 during a read operation.
For each chip, the write enable line is set to 1 during a write operation, and the output enable lines are set to 1 during a read operation.
Two-level addressing

Used to reduce the size of the encoder. Here is a diagram for a traditional 4M x 1 bit RAM

The decoder should have 4 million output lines. This makes the design complex.

In two-level addressing, the memory cells for any bit position are logically arranged as a two-dimensional array. 11 address lines are used to select a row, and 11 address lines are used to select a column. This reduces the number of output lines of the decoders from 4M to 2048 + 2048 = 4096 only.
**Example of two-level addressing**

A 4M x 1 bit memory. To save pins, in DRAMs, a set of 11 common lines are used for both A20-A11 and A10-A0. A pair of signals (RAS and CAS) are used to notify which part of the address is present at the input lines.
A “1” in the word line closes the switch, and the input data through the bit line charges the capacitor. During reading, the voltage on the capacitor is sensed through the bit line to determine if the content is a “1” or a “0”.

Periodic refreshing is done by reading the content, and writing the value back into the cell. An entire row is refreshed in one cycle.
Very high packing density, but slow (60-100 ns access time) DRAM is a good choice for the main memory of a computer.

**Cycle time**

![Diagram](image)

Read/write  read/write  read/write

Smallest time between consecutive read/write operations.
**Speeding up DRAMs**

*Page mode DRAM*

The row address does not change - only the column address changes. Latest product is EDO (Extended Data Out) allowing an access time of 25 ns within a row.

*Nibble mode DRAM*

Internally generate the next three column addresses.

*SDRAM (Synchronous DRAM)*

Series of bits from a row are automatically transferred in a burst - explicit column addresses are not specified.
Error Detection and Correction

Transmission or reading errors can corrupt data. If the extent of the damage is known, then error detection codes can detect if the data has been corrupted. Using special type of error-correction codes, we can even correct errors.

Parity Checking for single error detection

The value of P is chosen so that the number of 1’s in the transmitted data is odd (or even). The parity checker checks this in the received data.
Error Correction code

ECC can not only detect if there is an error, but also locate where the error is, and can correct it. Here is an example:

```
  1 0 1 1 1 1 0
```

The 4-bit data (1 0 1 1) to be sent is placed in bit positions 7, 6, 5, 3 of a 7-bit frame. The remaining bit positions are reserved for error-correction bits, and their values are chosen so that there is odd parity for bits (4, 5, 6, 7), (2, 3, 6, 7), and (1, 3, 5, 7).
The receiver, upon receiving the 7-bit data, computes the parities of the above bit combinations, and reports the result using three bits b2, b1, b0 as follows:

Odd parity for bits 4, 5, 6, 7 ⇒ b2 = 1 else b2 = 0.
Odd parity for bits 2, 3, 6, 7 ⇒ b1 = 1 else b1 = 0.
Odd parity for bits 2, 3, 6, 7 ⇒ b0 = 1 else b0 = 0.

If b2 b1 b0 = 000 then there is no error, otherwise, the decimal equivalent of b2 b1 b0 reports the position of the corrupt bit. To correct the error, the receiver flips that bit.

Works for single errors only. Try it out!
Can you figure out why it works?
Design using ROMs and PLAs

**ROM = Read Only Memory**

You are familiar with CD ROM. This is not a random access device, and is an electro-mechanical device.

We are interested in a ROM that is a solid state device. Its contents can be randomly accessed, very similar to RAM, with the exception that it contains fixed codes that cannot be erased or overwritten by the programmer.

Needs special equipment to write data into it. Some can be programmed only once, and often done by the manufacturer. In the programmable variety of ROMs (PROM), the contents can be erased (some can be erased by ultraviolet radiation) and reprogrammed.
Examples

Flash Memory is very popular. It is a form of EEPROM (Electrically Erasable Programmable ROM). Extensively used in your PC’s BIOS.

**SmartMedia** and **CompactFlash** are known as electronic films in digital cameras. Other products include Sony’s **memory stick**.
Design using ROMs

Example 1. Design a full adder using a ROM

\[
\begin{array}{ccc}
A & B & C \\
\hline
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

The content of the ROM are given in the table below:

<table>
<thead>
<tr>
<th>address</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 2
Can you design a binary mod-8 up counter using a ROM? 
Yes. Discussed in the class.

Example 3
Can you design an ALU using a ROM? Yes. 
Discussed in the class.

Internal structure of a ROM. An illustration using the 
design of a full adder using a ROM.
Programmable Logic Array (PLA)

In a ROM, the AND section is a decoder that generates all the $2^n$ outputs. The OP section can be programmed according to our design needs.

In a PLA, both the AND section and the OR section can be programmed. By programming the AND section, we generate only those boolean product terms that we need.

Any combinational circuit can be designed using a PLA. Any sequential circuit (therefore, any control unit) can be designed using a set of flip-flops and a PLA. How?
**Design of the MIPS processor**

We will study the design of a simple version of MIPS that can support the instructions

- I-type instructions LW, SW
- R-type instructions, like ADD, SUB
- Conditional branch instruction BEQ
- J-type branch instruction J

We will closely follow the material from Chapter 5 of the textbook. Follow the class lectures.
### The instruction formats

<table>
<thead>
<tr>
<th></th>
<th>6-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>SW</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>ADD</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
<tr>
<td>SUB</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>func</td>
</tr>
<tr>
<td>BEQ</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>address</td>
</tr>
</tbody>
</table>
**How to generate the ALU control input?** The control unit first generates a **2-bit ALU op** from the opcode of the instruction. This information is combined with the **function field** of some of the R-type operations to generate the **3-bit control input**.
Design of MIPS (continued)

Follow the design steps from the class lectures.

We will study the datapath, the control unit, and the performance of the simple version of MIPS that executes every instruction in one cycle. Read these from Chapter 5.

After this, we will study the more realistic version of MIPS, the multi-cycle version.