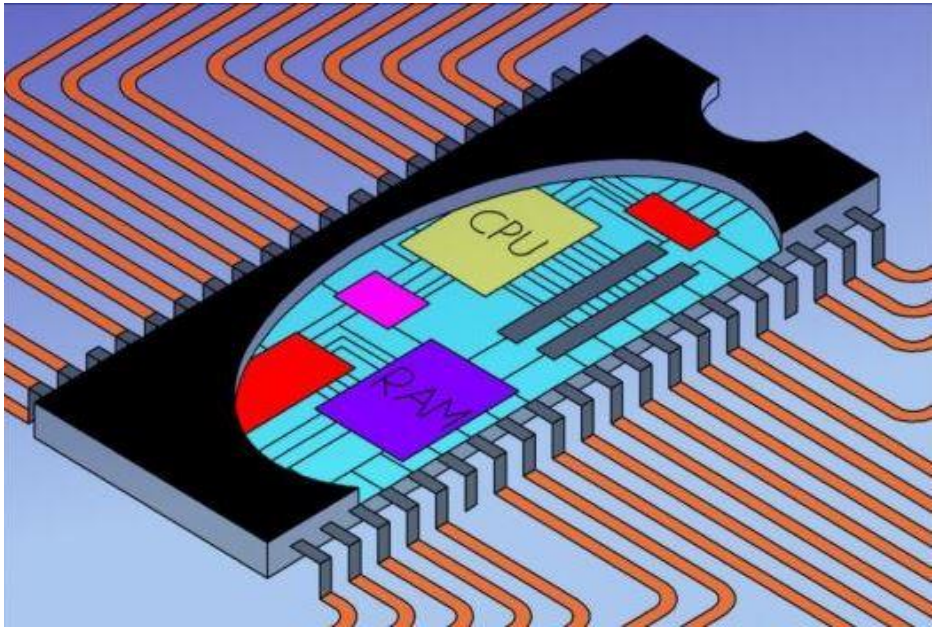
The University of Zambia

EEE 3132 – DIGITAL ELECTRONICS



Jasper HATILIMA

Department of Electrical and Electronic Engineering, School of Engineering

jasper.hatilima@unza.zm
jasperhatilima@yahoo.com

January 2015

Course Outline

Jerry Muwamba

Number Systems and Codes

Logic Fundamentals

Jasper Hatilima

Logic Families and Combinational Logic Circuits:

- Logic Families, TTL, Binary adders and subtractors, Encoders and Decoders, Multiplexers and Demultiplexers.

Sequential Logic Circuits:

- S-R latch, Clock S-R Flip Flop, Level and Edge Triggering, J-K, D-Flip Flops, State Tables, State Diagrams.
- Serial/Parallel In/Out shift registers.
- Timing diagrams, Asynchronous (unclocked) and synchronous systems, Counters: ripple, synchronous, ring counters.

Introduction to Microprocessors/Microcontrollers and PLDs:

- Three State Registers.
- Memories: ROM, PROM, EPROM, EEPROM, SRAM, DRAM.
- Microprocessor Architecture, Instruction Set, Assembly language

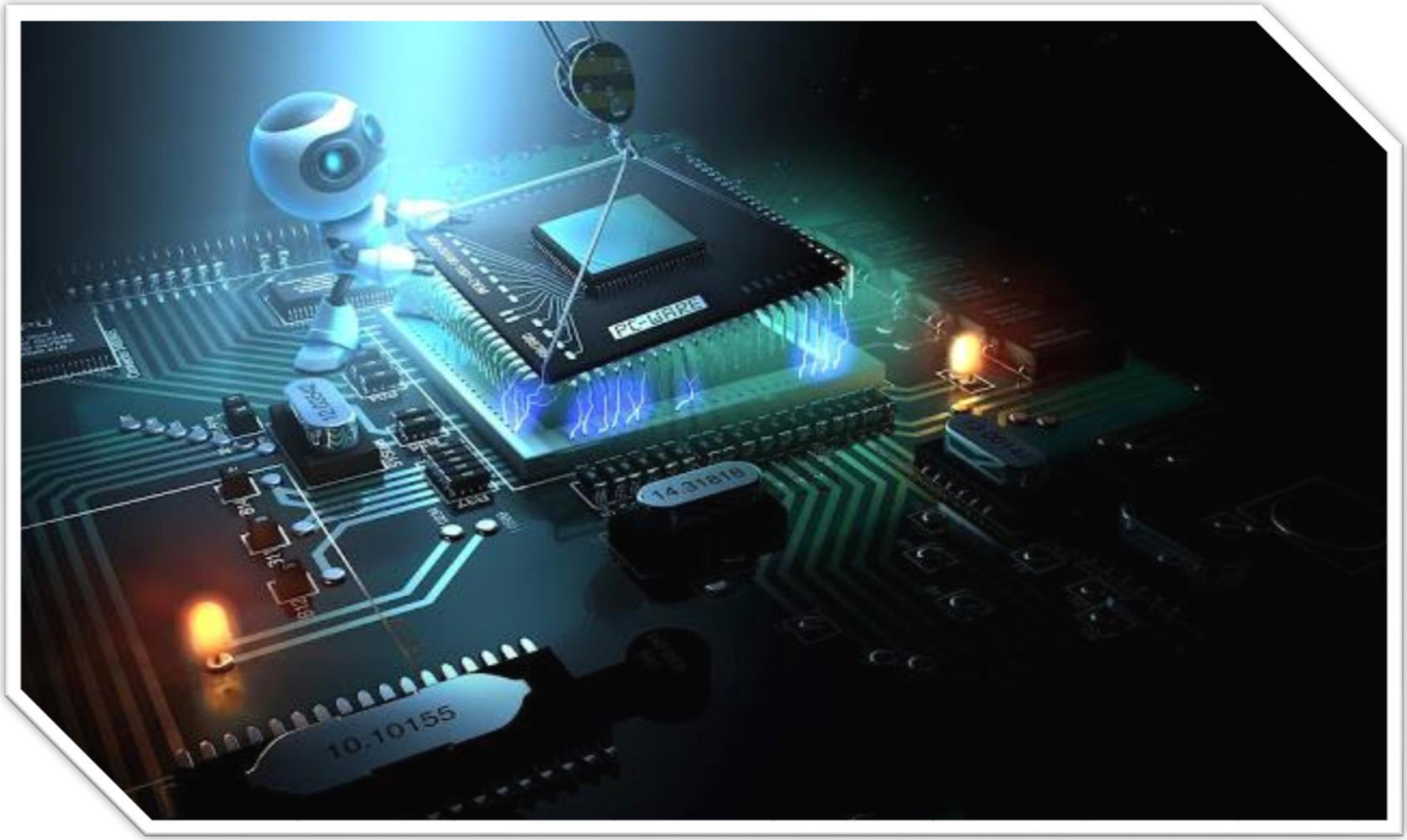
PART 3

Intro. to Microprocessors and PLDs

At the end of this component, the student should:

- be able to describe the principle behind volatile and non volatile storage.
- understand Programmable Logic Device concepts.
- appreciate microprocessor architecture and be able to design a very simple microprocessor based computer.
- understand the basics of programming with an emphasis on assembly language.

INTRODUCTION



- We started with discrete components: diodes, transistors, etc.
- Then we integrated them into ICs using TTL, ECL, CMOS
- Along the way we gave examples of specific TTL IC implementations of various sub-systems: combinational and sequential logic. Here we will add a few more TTL implementations of memories.
- And then we will further combine the various sub-systems that we have learnt into more practical systems that can do meaningful tasks.
- One such system is a digital computer. We also need to go to a high level by introducing computer programming: A method of human-computer interaction.

PART 3.1
THREE-STATE BUFFER REGISTERS AND MEMORIES

Refer to Figure 2.21 under buffer registers. It is repeated in Fig. 3.1.1 below showing the TTL implementation of a 4-bit controlled buffer register (74LS173)

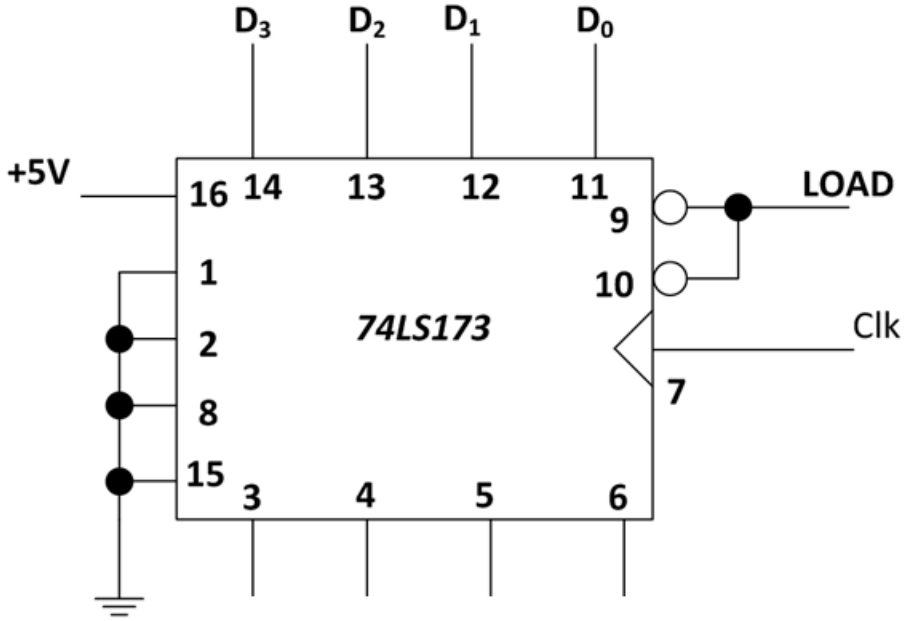


Fig. 3.1.1

Notice Pins 1 and 2 that are used for controlled output.
How is this implemented?

3.1.1 Three-State Switch (Tri-state switch)

The output has three possible states: LOW, HIGH, or Floating also called high-impedance.

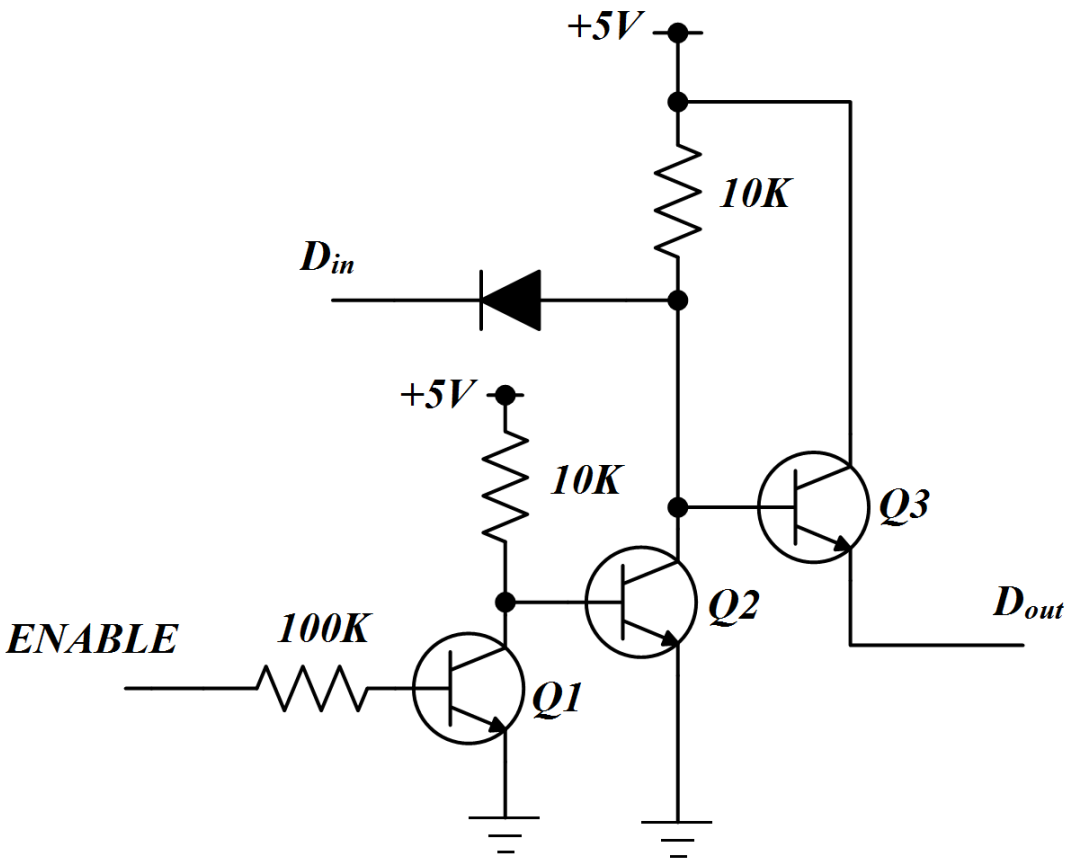


Fig. 3.1.2 Normally Open Three-State switch.

Operation:

When ENABLE is LOW, transistor Q1 cuts-off and Q2 saturates thereby cutting-off Q3. This makes the output to float.

When ENABLE is HIGH, transistor Q1 saturates and consequently cuts-off Q2 which in turn saturates Q3. This is equivalent to a closed switch i.e. the value of D_{in} is equal to D_{out}.

The Tri-state switch can also be normally closed as shown by the logic symbol in Fig. 3.1.3 (b) below:

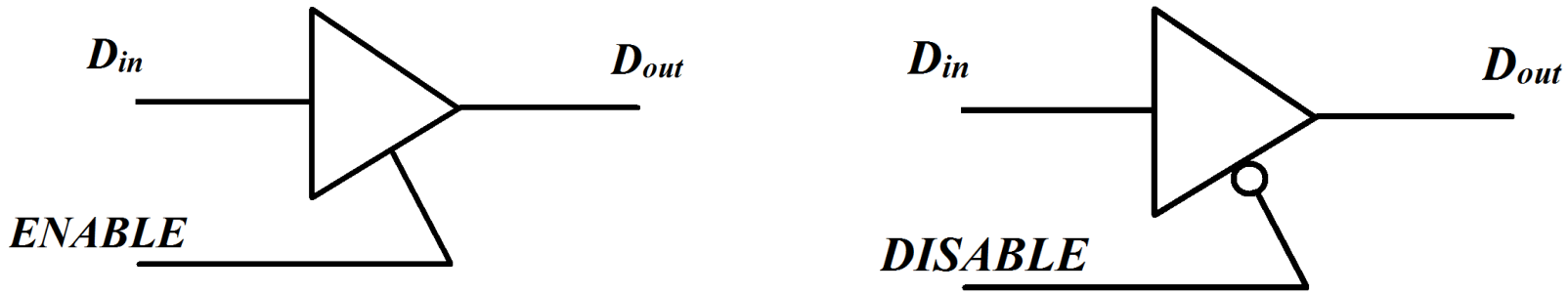


Fig. 3.1.3 (a) Normally Open (b) Normally Closed

TTL implementation:

The Three-state (Tri-state) switch is implemented on an IC as **74126** which is shown below as having four tri-state switches.

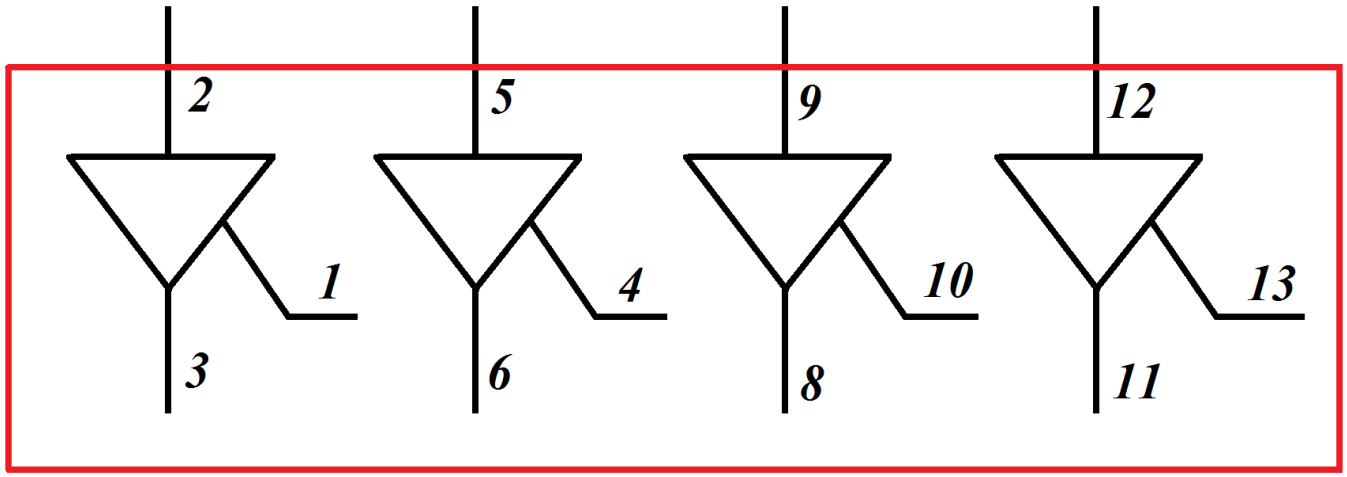


Fig. 3.1.4 TTL (74126) Tri-State switch

Application of Tri-state switch:

The Three-state switch is used to create a three-state buffer register from a controlled buffer register of **Fig. 2.20** under sequential logic component.

In **74LS173** of **Fig. 2.21**, the tri-state switch is incorporated into the IC and controlled by pins 1 and 2.

If the ENABLE is low, the output pins are open (high impedance) and so the contents of the buffer register are isolated from the **bus**.

*This brings in the concept of a bus organised computer that will be discusses later. This has a **Data Bus**, **Control Bus** and **Address Bus**.*

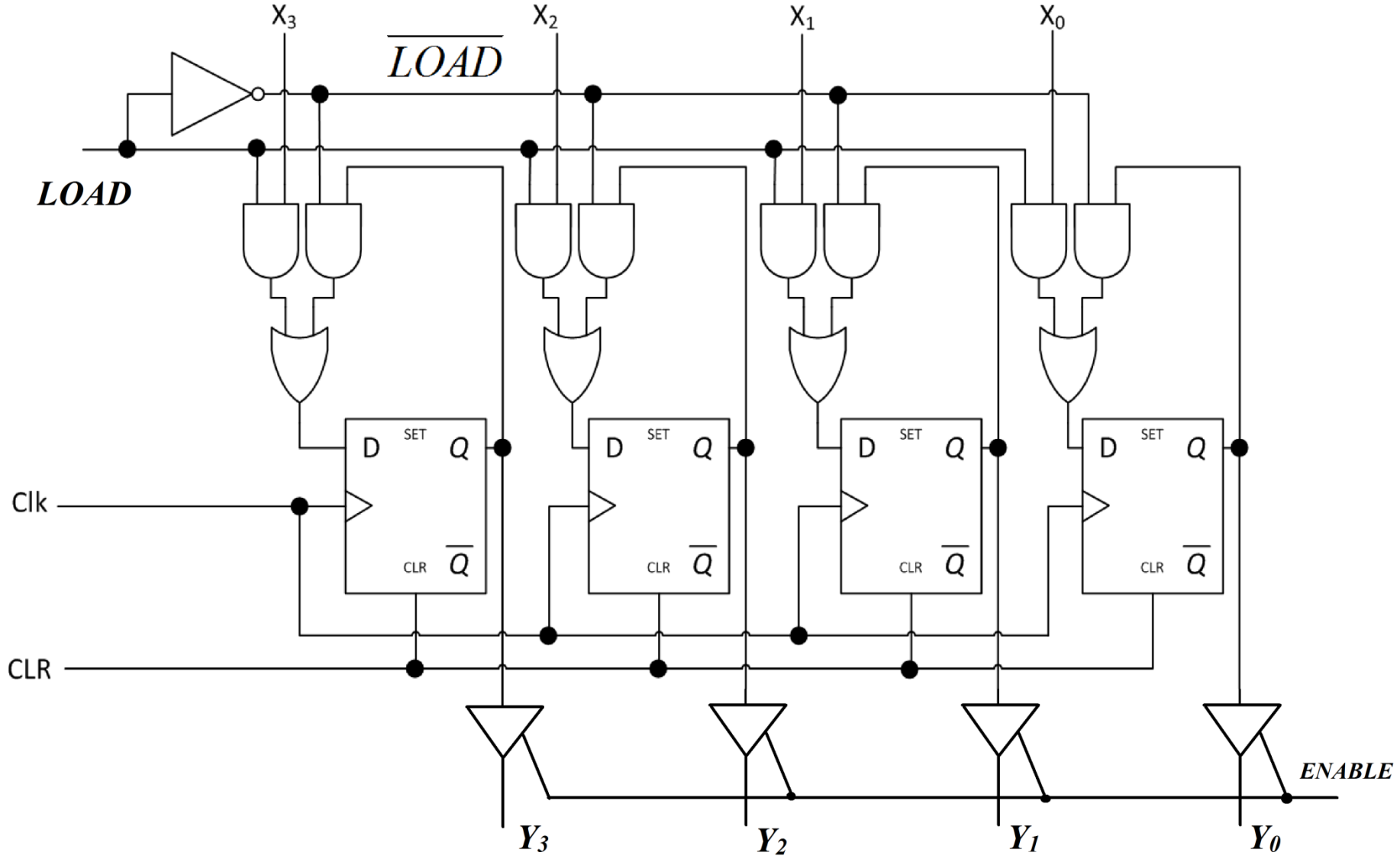


Fig. 3.1.5 Three-State Buffer Register

3.1.2 MEMORIES

✓ Computer systems need memory to store the program and data.

✓ The memory is made up of millions of registers.

✓ Memories can be classified as volatile and non-volatile.

✓ And so in our discussion, we will first start with non-volatile memories.

✓ Then we will put together the basic concepts of latches that we have already done so as to better understand volatile memory.

At this point, it is necessary to understand **MEMORY ADDRESSING.**

So before we go further, let us see what we have done so far and how it drives us towards a functional system

- ✓ Number Systems and Boolean Algebra
- ✓ Logic Implementation and logic families.
- ✓ Combinational and Sequential logic circuits.

Refer to your computer for example, how does the above apply?

I will make reference to what we have covered in the following order:

1. **Binary Numbers**: Computer process info using a signal's HIGH or LOW and binary numbers are a perfect abstraction for this.

2. **Hexadecimal Numbers**: Compress binary numbers.

3. **Combinational logic**: Arithmetic, decoding, bus control etc

4. **Sequential Logic**: Registers (Memory), Counters, etc

5. **Computer Memory** is becoming complex and therefore one way of addressing it during Human-Computer interaction is using HEX numbers.
6. **Addressing the large computer memories** requires address decoders.

Expressed diagrammatically…

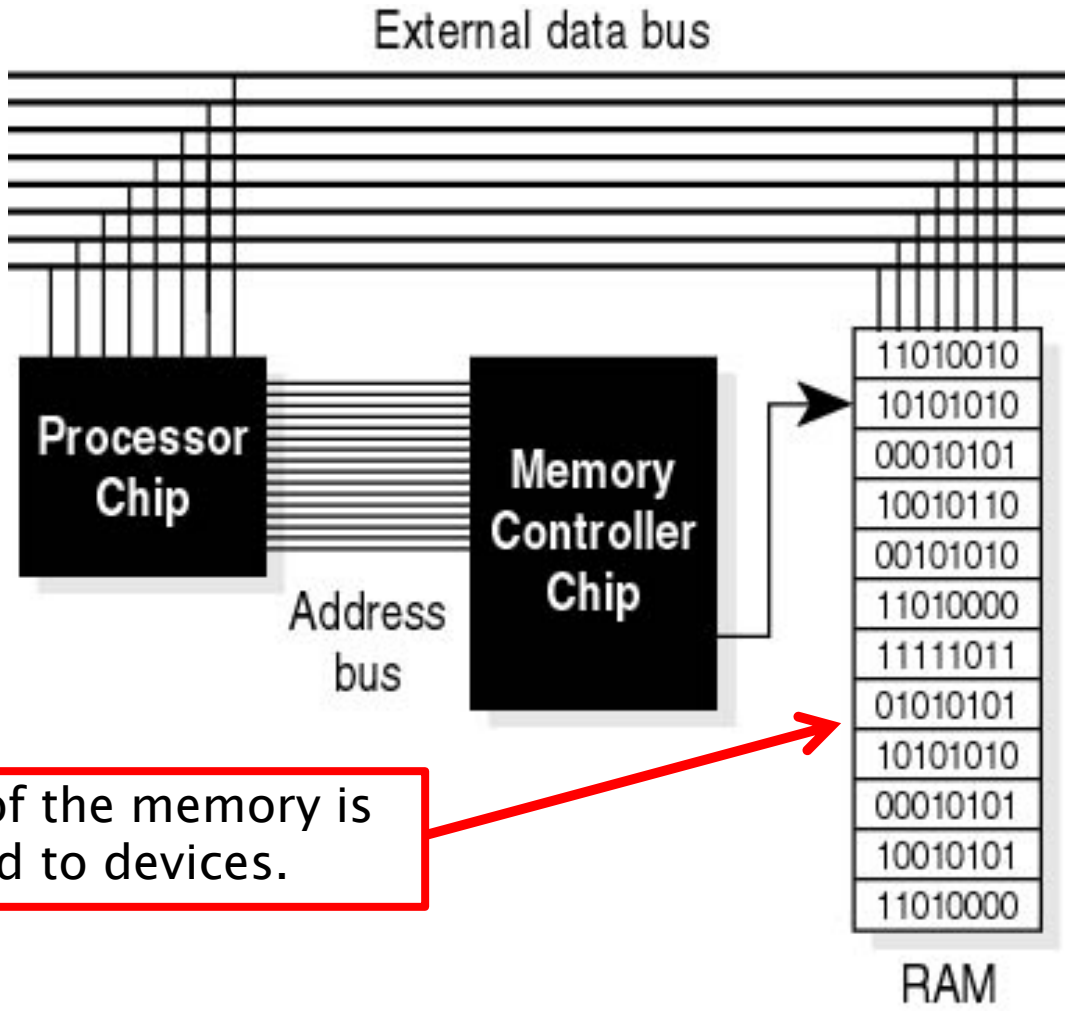
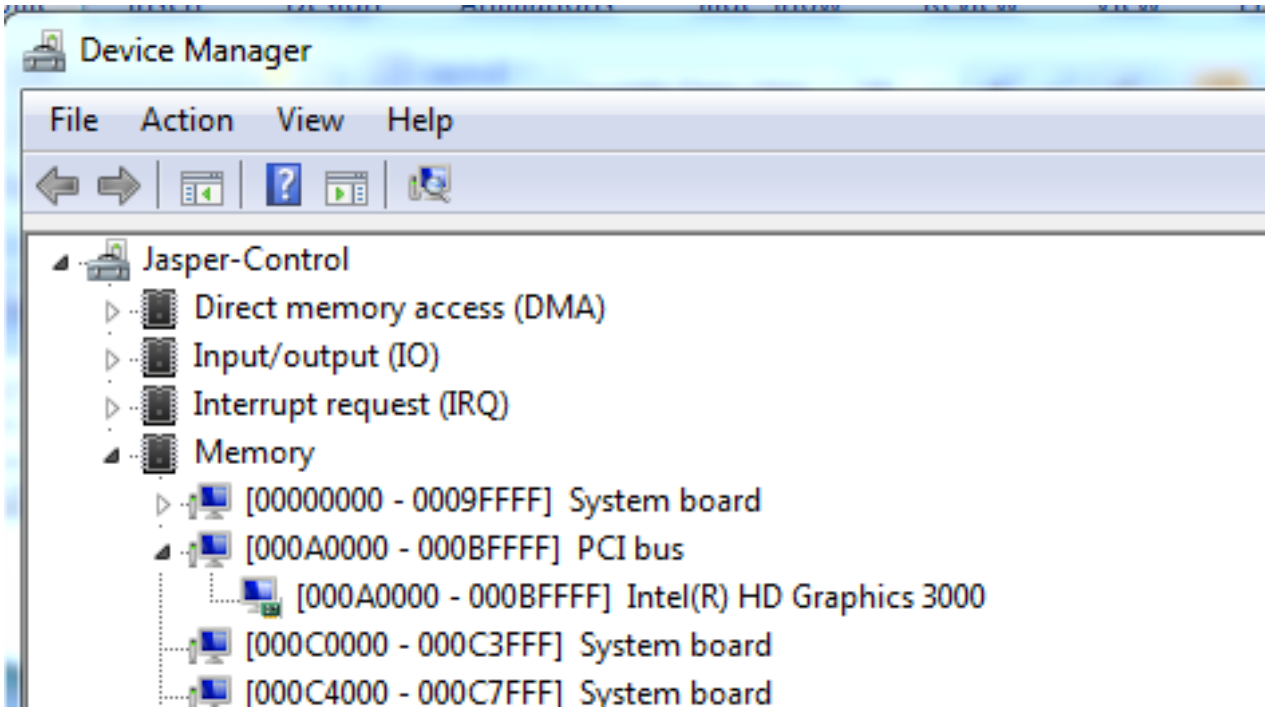


Fig. 3.1.6 Digital Computer System

Memory is a key part of a computer system. We therefore need to understand MEMORIES (RAM, ROM, etc)

- Go to “Device Manager” or you can search for “devmgmt.msc”
- Go to “View” then “Resources by Connection”
- Expand the “Memory” Node.
- Go ahead and expand the “PCI bus” nodes to see the ranges and the devices consuming the mapped memory.



In the next slides, we will start with non-volatile memories and examine their implementations.

1. We define **non-volatile memory** is the kind that retains the stored contents even after the power to the device has been switched off.

2. On the other hand, **volatile memory** is the kind of memory that does loses the stored values once power to the memory is switched off.

NON-VOLATILE MEMORY

1. ROMs

✓ Read-Only-Memory is a group of registers with data permanently stored.

✓ We need to apply appropriate control signals in order for us to read data from the ROM.

✓ Used to store information closely tied to specific hardware. E.g. Diode ROM below (in fabrication BJT or MOSFETs are used).

The register contents for the diode ROM are shown below:

Register	Address	Contents $D_3D_2D_1D_0$
REG0	0	1000
REG1	1	1011
REG2	2	0110
REG3	3	1110

Implementing the Diode ROM using switch-select addressing:

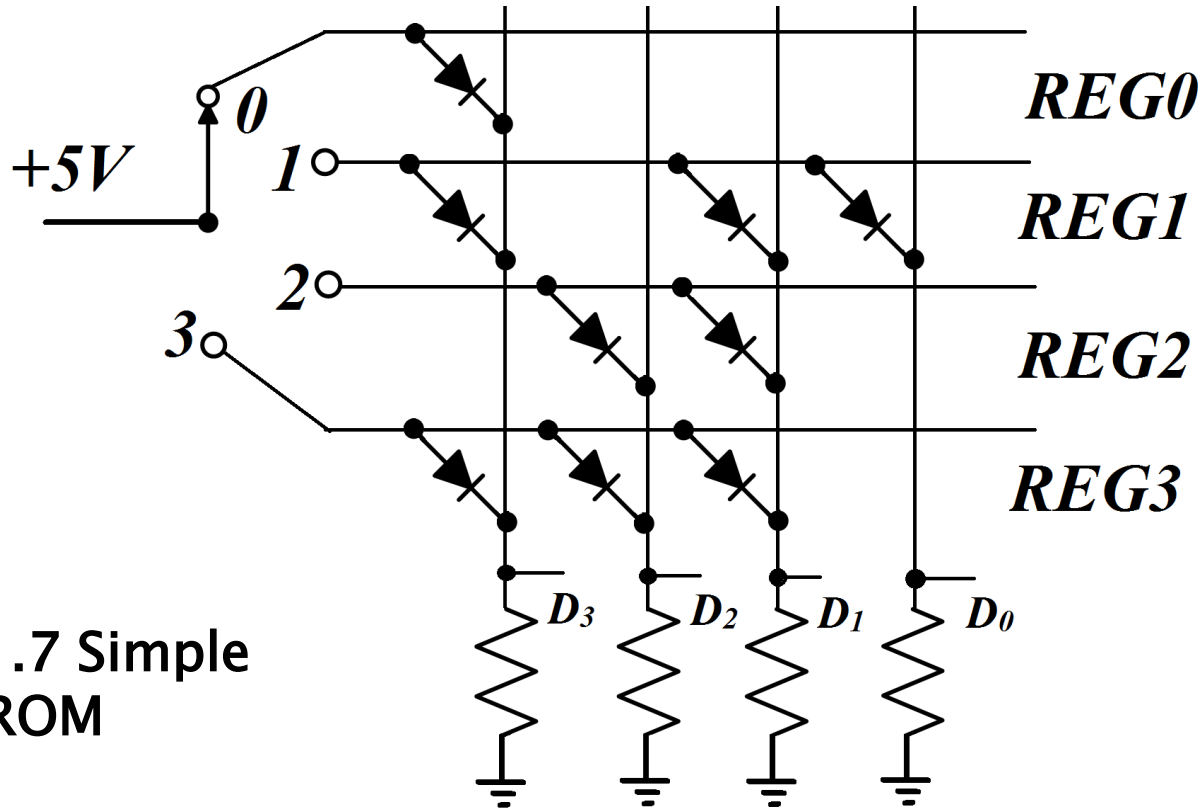


Fig. 3.1.7 Simple Diode ROM

0, 1, 2 and 3 are called **Word-Lines** because they address the whole word (4-bits in this case). The D's are called **Bit-Lines** because they are connected to 1-bit at a time.

Notice that in the ROM above, four addresses are required for the four registers and therefore the switch must have four contacts to which it can be thrown.

But **recall that we can use a decoder** that gets for example 3 address lines (bits) and activates only one of the eight outputs (bits).

In this case, we can use a **2-to-4 decoder** to select a register (Memory location)

Decoder implemented on the chip, it is called **On-Chip Decoding**.

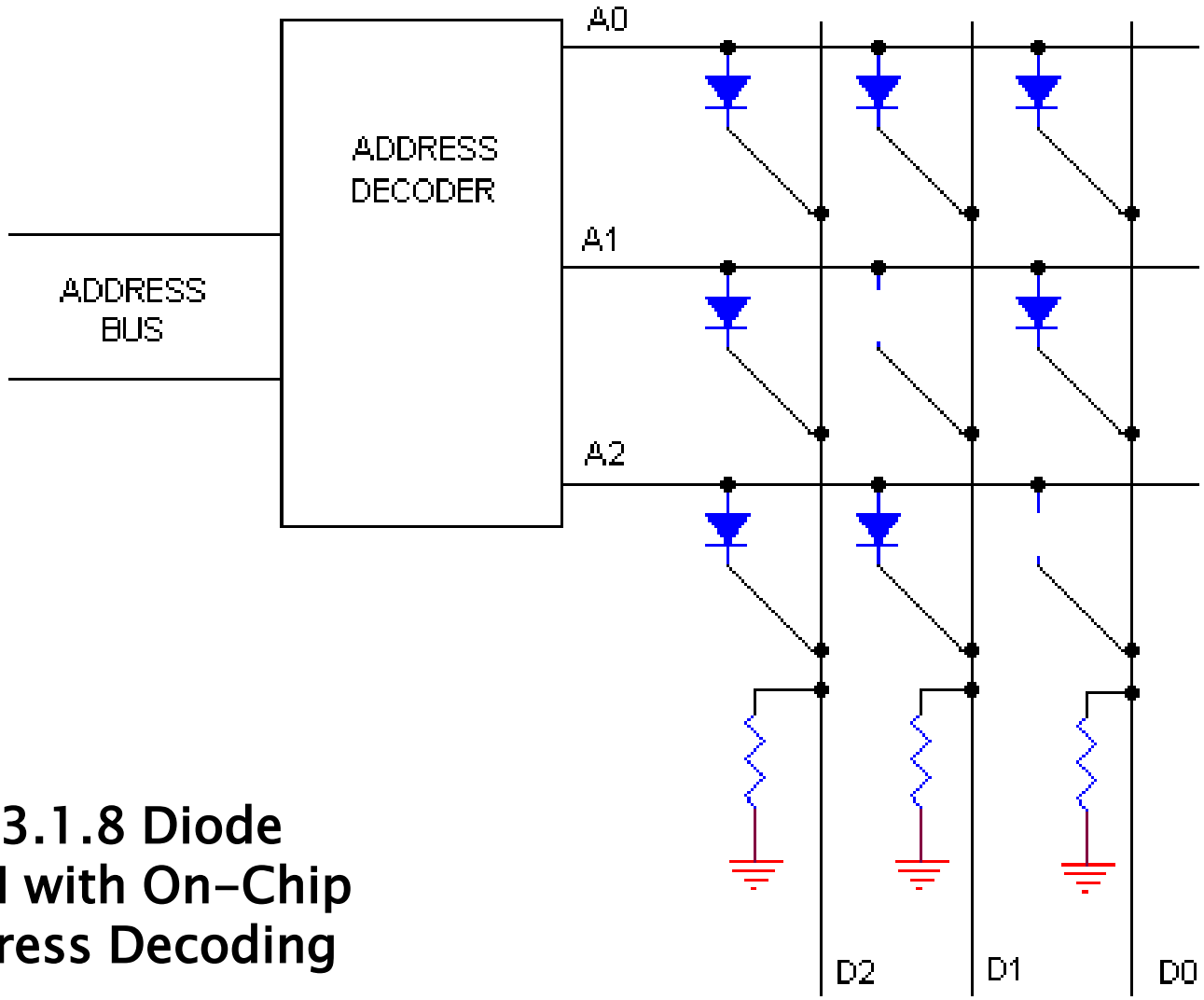


Fig. 3.1.8 Diode ROM with On-Chip Address Decoding

The A's are Word-Lines and the D's are Bit-Lines.

QUIZ 1:

Using simple On-Chip Decoding, how many memory locations on a ROM can be accessed on a system using 32-bit long addresses?

2. PROMs

✓ Programmable Read-Only-Memory (PROM) can be **programmed by the user**.

✓ The rows and columns of the memory grid are linked by **transistors and fuses**.

✓ By passing a charge through the column using a special 'burning' tool called a **PROM programmer**, the fuse connections for the grounded row are burnt open.

✓ This means that a **blank PROM initially has all 1's stored** and the 0's are programmed during the burning process.

✓ Once the data and/or program has been stored, it **cannot be erased**.

✓ PROMs **are fragile**: jolts of static electricity can burn some fuses.

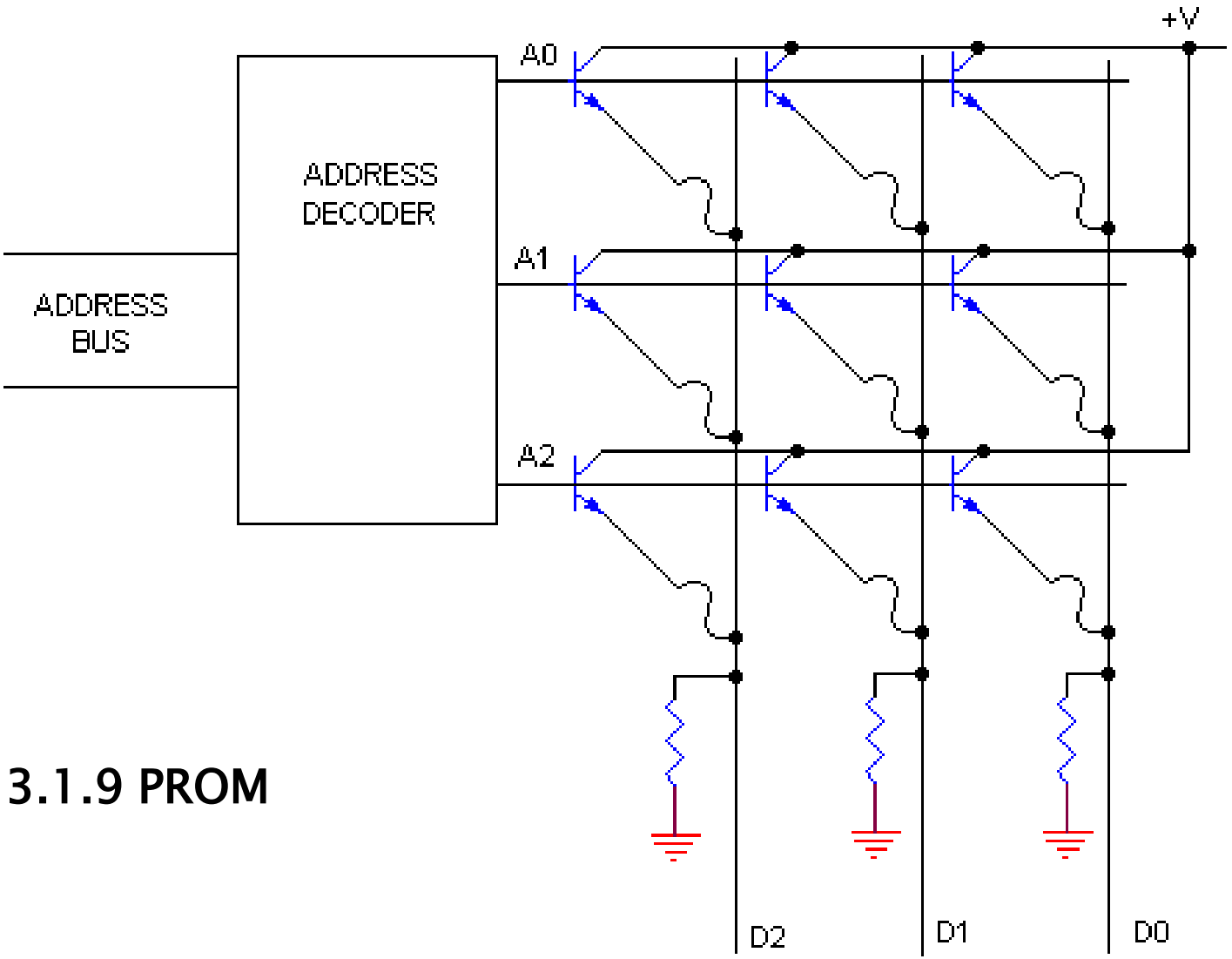


Fig. 3.1.9 PROM

3. EPROMs

✓ ROMs and PROMs can be inexpensive per chip but their cost adds up over time as you want to do something new...you have to buy a new one.

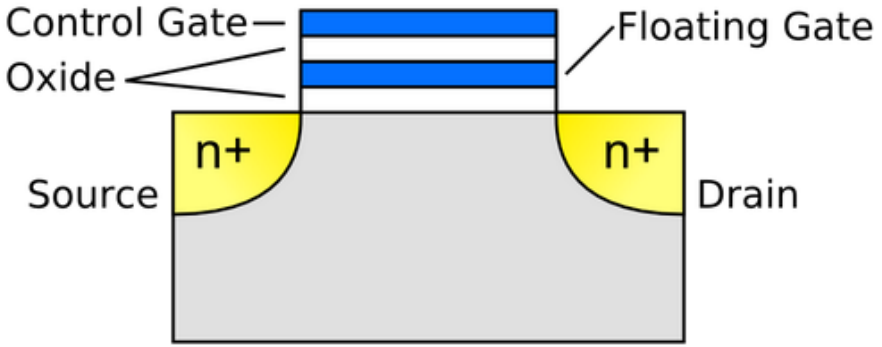
✓ Erasable Programmable ROM (EPROM) addresses this cumulative cost issue.

✓ EPROMs can be electrically programmed (written to) and then erased using ultraviolet (UV) light e.g. Mercury-vapour light.

✓ **ULTRAVIOLET** light passes through a window in the IC package.

✓ The rows and columns have a MOSFET at each intersection, and a floating gate FET as memory element (**this is one way of implementing**)

Fig. 3.1.10 Floating Gate FET



✓ To start with, we know that in a normal MOSFET current from **SOURCE** to **DRAIN** flows when **GATE** voltage is applied.

✓ In the Floating Gate FET, electrons can **TUNNEL** across the oxide between **Control GATE (CG)** and Floating Gate (**FG**) by applying a high pulse typically between 20-25V to V_{PP} terminal.

✓ A permanently charged FG implies that the transistor is ever conducting.

✓ The simplified diagram below shows MOSFET/FG FET in EPROM

Just one of the ways of implementing EPROM. Key is MOSFET and FG-FET

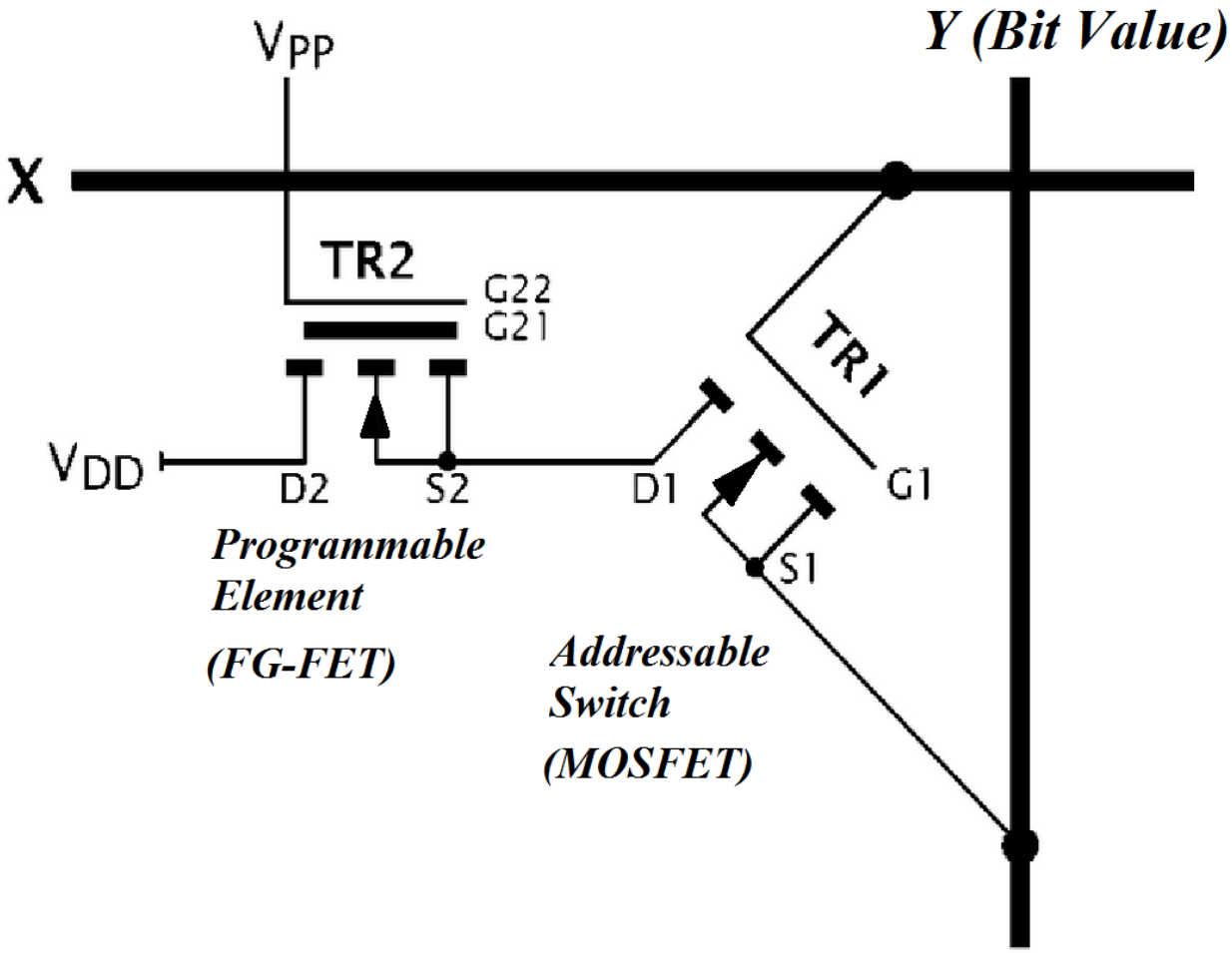


Fig. 3.1.11 A simple EPROM Cell

✓ The MOSFET replaces the diodes in ROM as cross-grid devices.

✓ The gate of the MOSFET is connected to the X line (**Word-Line**) and the Source is connected to the Y line (**Bit-Line**).

✓ MOSFET will only conduct when FG-FET is conducting.

✓ FG-FET will only conduct when its CG/FG combination (i.e V_{PP} + FG charge) has a resultant positive threshold voltage.

✓ Therefore **if FG was charged positive after tunneling**, FG-FET will always conduct and so will MOSFET => Logic 1 will be read on Y line.

✓ If **FG was not charged**, FG FET will not conduct and MOSFET will not either => Logic 0 will be read on Y line.

Erasure from an EPROM is done by passing UV light through a transparent window on the IC. The UV light passes its energy to the electrons on the FG and they break away.

The EPROM is taken off the circuit into a UV EPROM eraser.

This erasure happens to the entire memory.

Fig. 3.1.12 EPROM IC

4. EEPROMs

✓ Electrically Erasable PROMs (EEPROMs) do not use UV light to erase.

✓ This means that they **can be erased while still installed** in the system by using a negative pulse on V_{pp} of the EPROM of figure 3.1.11. This makes the electrons to tunnel back.

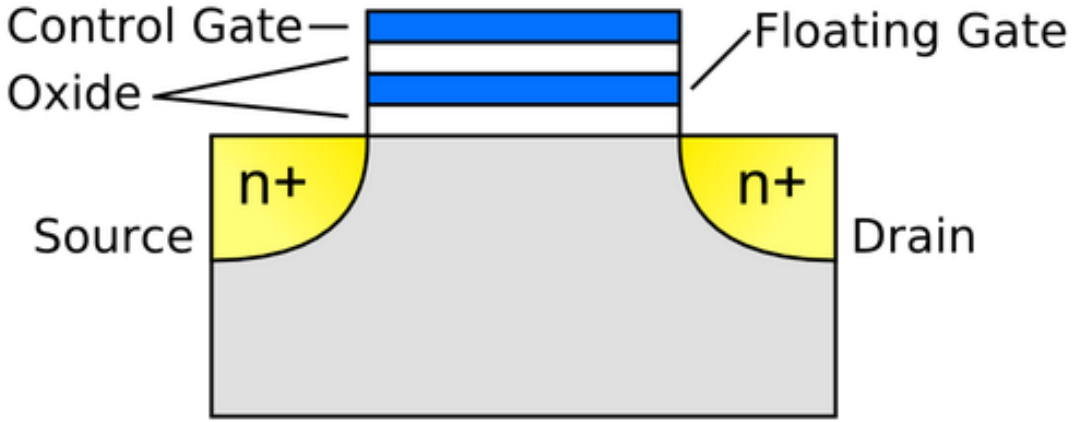
✓ This negative voltage is generally generated on the chip and therefore EEPROM has the following advantages over EPROM:

I. The chip does not have to be removed to be written/erased.
II. The entire chip does not have to be entirely erased just to change a specific portion of the contents. The erasure is done byte-by-byte.
III. Erasing does not require special equipment like UV machine.

5. FLASH MEMORY

✓ The term '**FLASH**' was chosen because the entire memory can be erased in a flash unlike EPROM which takes minutes in UV or EEPROM which erases one byte at a time (slow).

✓ The basis for flash memory is still Floating Gate FETs which we refer to as FG-FETs in this presentation.



✓ In the Floating Gate FET, electrons can be channeled from the drain to the floating gate (FG) by giving them enough energy to cross the oxide silicon barrier – **Channel Hot Electron** (**CHE**), sometimes **FN Tunneling** can be used.

✓ Once these electrons are trapped on the FG, they can be used to modulate the Control Gate (CG) voltage and therefore increase the threshold required to activate the channel as shown below.

✓ We will not go into the detailed physics but just know that a MOSFET has a **threshold voltage, V_T** ,that has to be applied to the Gate at which the Channel will conduct a certain amount of current from SOURCE to DRAIN.

✓ The same V_T applies to the Floating Gate FET as described in simple terms below.

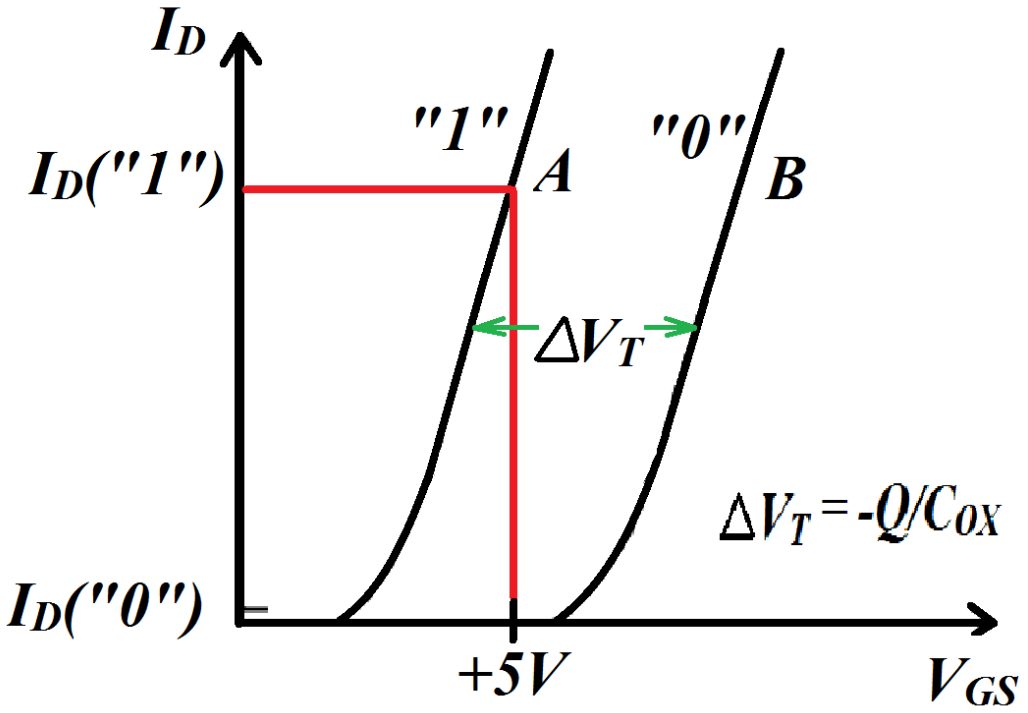
For a Floating Gate FET, the threshold voltage is given by

$$V_T = K - Q/C_{ox}$$

Where **K** is a constant depending on gate and substrate material. **Q** is the charge stored in the floating gate and C_{ox} is the coupling capacitance between FG and CG.

So notice that with electrons (negative charge) stored on the floating gate, the threshold voltage to achieve a certain amount of current flow in the channel will increase.

Fig. 3.1.13 Reading from FG FET



A cell sensor can then be used to measure the current passing and correspondingly decode as a 1 or a 0.

ERASURE is implemented using **FN tunneling** (Fowler-Nordheim) to remove the charge from FG. This implies that blank FLASH memory is all 1's (no charge).

It would be interesting to do personal reading on

✓ NOR flash and NAND flash.

✓ Also read about ATA (Intelligent Controller) verses Linear Flash Cards (Flash File System Software).

✓ Bootloader (BIOS) is a form of ROM…though in new machines it is being implemented as Flash Memory to allow firmware upgrade.

✓ We have studied non-volatile memory. ROM, PROM, EPROM, EEPROM and Flash Memory.

✓ We now discuss volatile memory and how it is implemented.

Any known examples of areas where volatile memory is applied?

VOLATILE MEMORY

RAM

✓ Random Access Memory

✓ Random Access in that data can be read and written to locations in the same amount of time regardless of physical location.

✓ This is in contrast to CDs, Hard disks etc where mechanical limitations like arm speed and data physical location can affect access speed.

✓This is a specific application of volatile memory.

✓ By studying this specific application, we will appreciate the general implementation of volatile memory.

Semiconductor RAMs are mainly classified under the following:

1. Static RAM (SRAM)

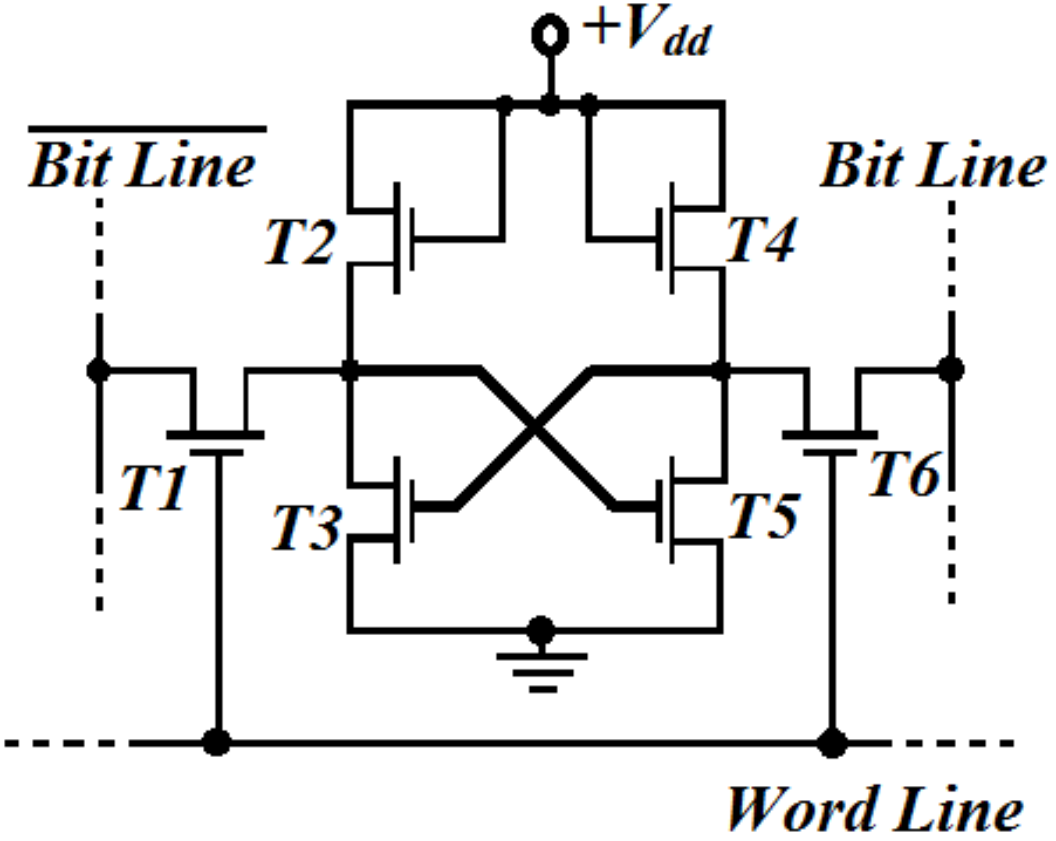


Fig. 3.1.14 a 6T Static RAM cell

✓ Uses BJT or MOSFET flip-flops to store data (or sometimes MOSFET inverters).

✓ This means that data is retained for as long as the biasing power is applied.

✓ The SRAM shown above is a 6 transistor (6T) SRAM. It can be converted to a 4T SRAM if T2 and T4 are replaced with resistors.

✓ T3 and T5 are the flip-flop **access transistors**; while T2 and T4 are **active loads**.

✓ The Bit-Line (BL) and the Word-Line (WL) are used to read (write) from (to) the cell.

✓ It is not always necessary to have ~BL, the complement of BL, but having them both improves the noise margin.

✓ In standby mode, the WL is low turning-off the access transistors.

✓ During read access, WL is high and then the two bit lines are actively driven HIGH and LOW by the flip-flop. This data is then sensed on BL and ~BL.

✓ During a write process, information (bit) is imposed on the BL and its complement on ~BL. Then the access transistors are turned on by the WL.

✓ As soon as information is stored in the memory cell, the access transistors are turned off.

2. Dynamic RAM (DRAM)

✓ The simplest form uses a capacitor and a transistor (1T) to form a cell as shown in figure 3.1.15.

✓ A 3T implementation is also common.

✓ The data is stored on the capacitor and therefore this requires refreshing every few milliseconds…typically 2ms.

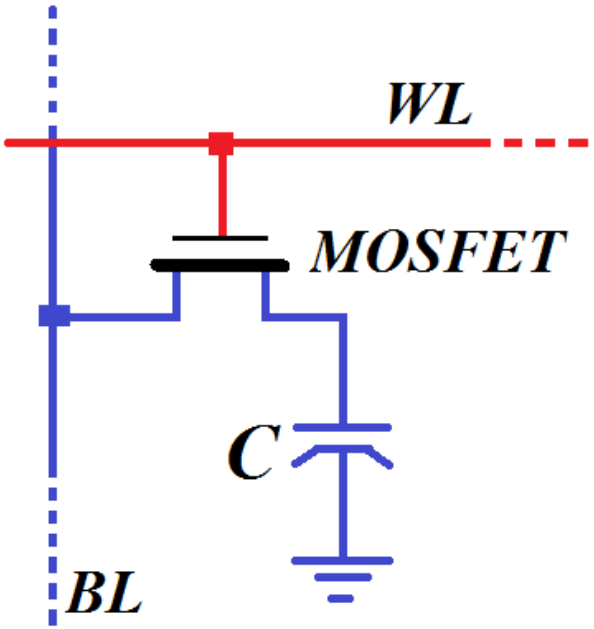


Fig. 3.1.15 a 1T Dynamic RAM Cell

✓ To read from the cell, the BL has sense amplifiers that read the charge on the capacitor to determine if the stored value is a 1 or a 0.

✓ To write to the cell, the WL is kept high and then the sense amplifier on the BL inputs the desired value HIGH or LOW thereby charging or discharging the capacitor.

✓ Below is a 4-by-4 cell matrix of DRAM.

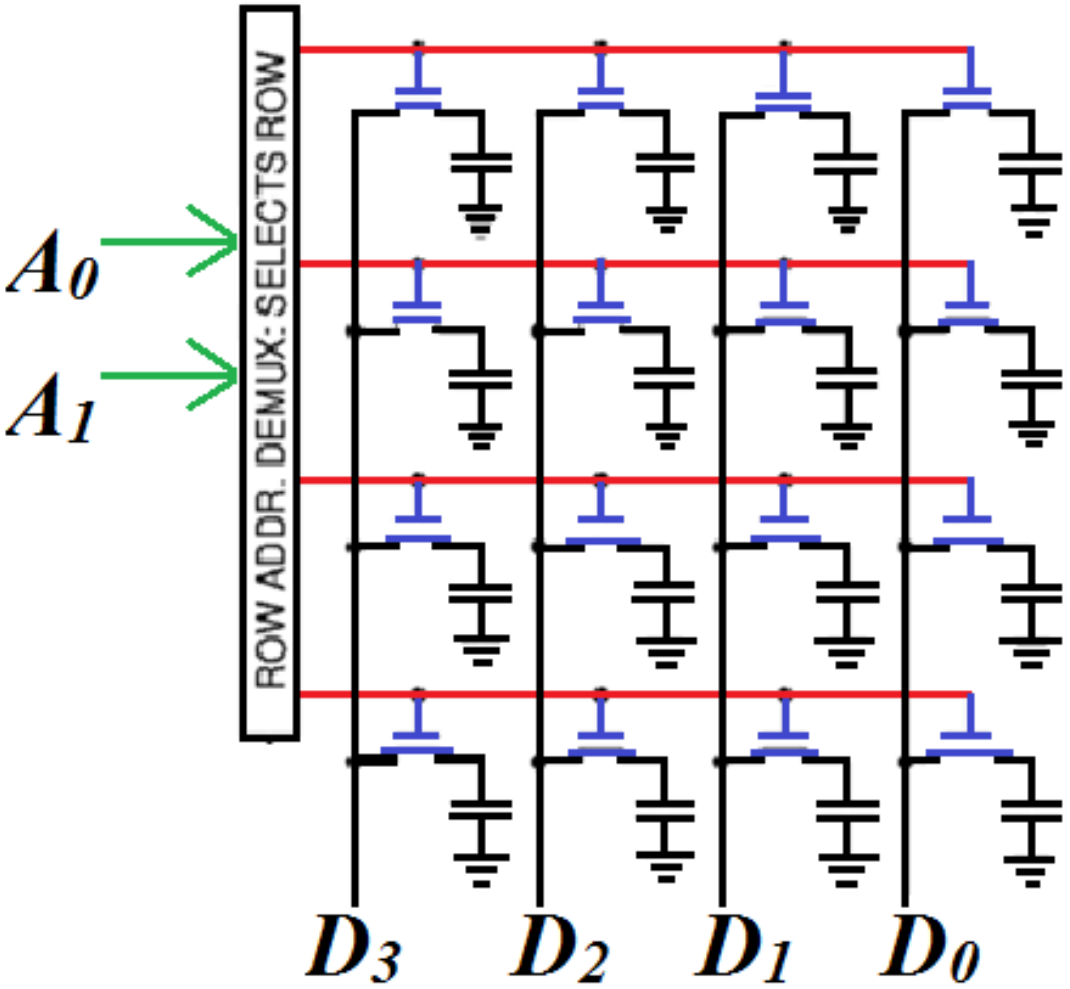


Fig. 3.1.16 a 4-by-4 DRAM Matrix

DRAM can also be implemented using a 3T configuration as shown below.

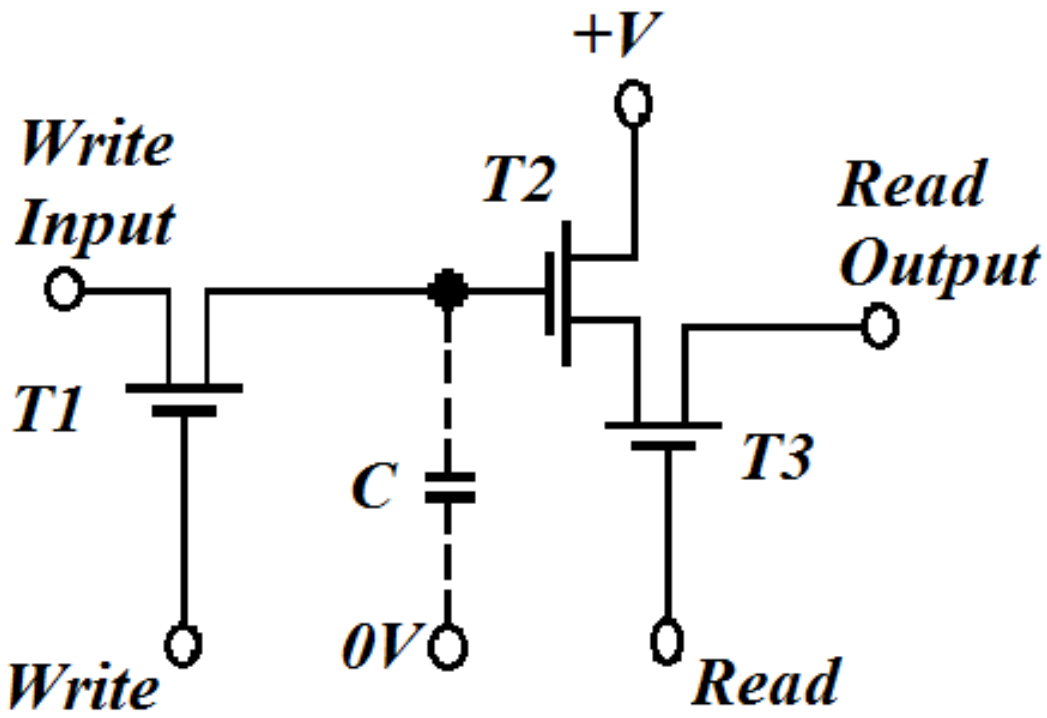


Fig. 3.1.17 a 3T Dynamic RAM Cell

From the discussion on SRAM and DRAM above we can see that:

✓ DRAM has more memory cells per unit area than SRAM.

✓ DRAM requires additional circuitry to refresh the capacitor.

✓ Since DRAM writes by charging a capacitor, it is much slower than SRAM.

Due to the above observations,
SRAM finds use in CPU cache where speed is needed and size is not really an issue.

DRAM is used as the main memory where large volume is needed. It is implemented as Synchronous DRAM (SDRAM) in modules we use such as DDR1, DDR2 and DDR3.

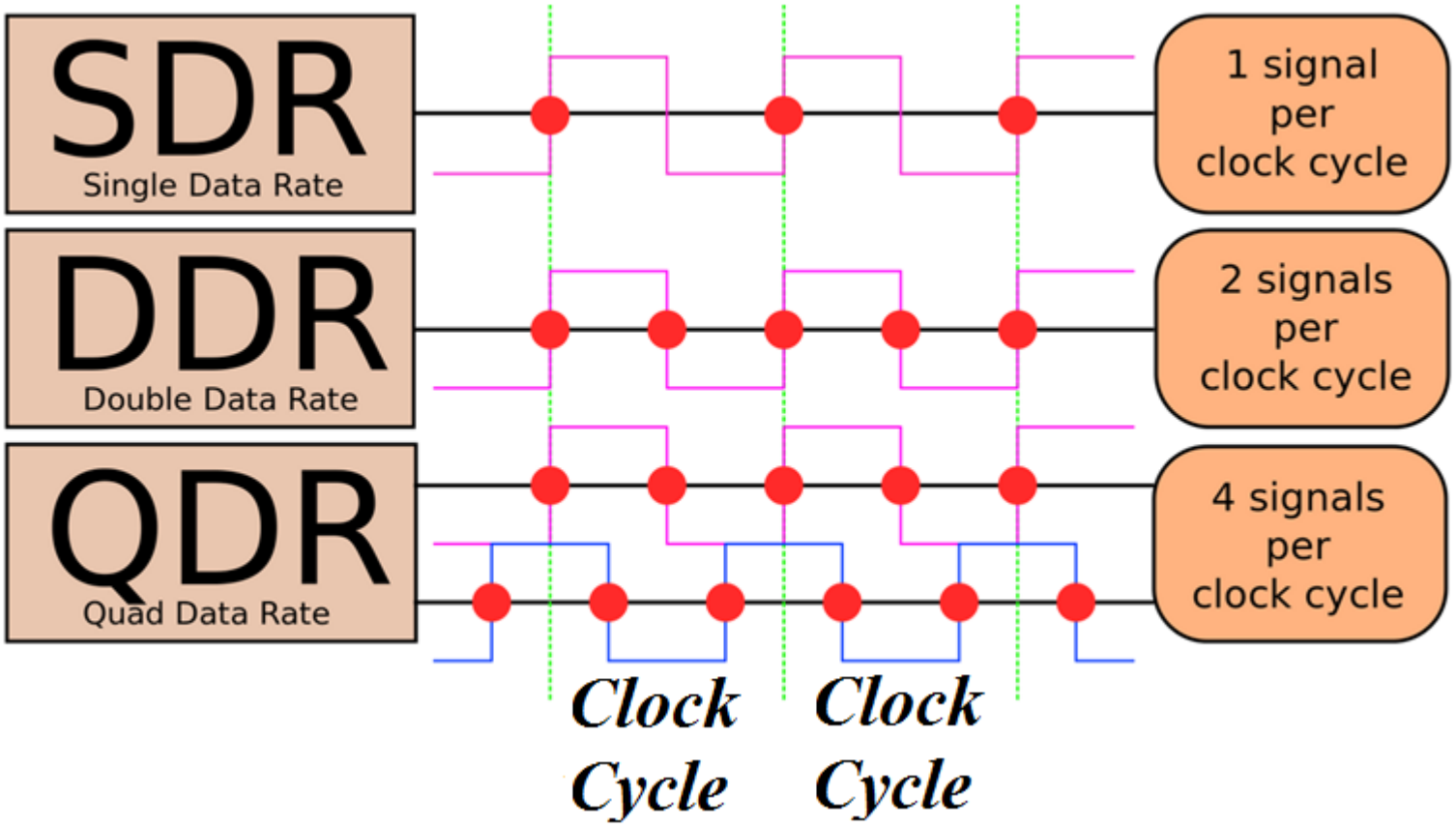


Fig. 3.1.18 RAM Clocking

Source: Wikipedia

PART 3.2
Microprocessor/Microcontroller Architecture

Difference between a microprocessor and a microcontroller?

The microprocessor (µP) is basically a Central Processing Unit (CPU) on a chip.

CPU has the Arithmetic and Logic Unit (ALU), Some registers to hold data being processed and a Control Unit.

On the other hand, a microcontroller (µC) is a CPU (microprocessor) with other useful components integrated on the SAME CHIP.

These other useful components that are integrated include:

- ✓ memory (Flash, RAM, EEPROM),
- ✓ timers,
- ✓ Analog-to-digital converters,
- ✓ built-in serial interface and other ports,

etc

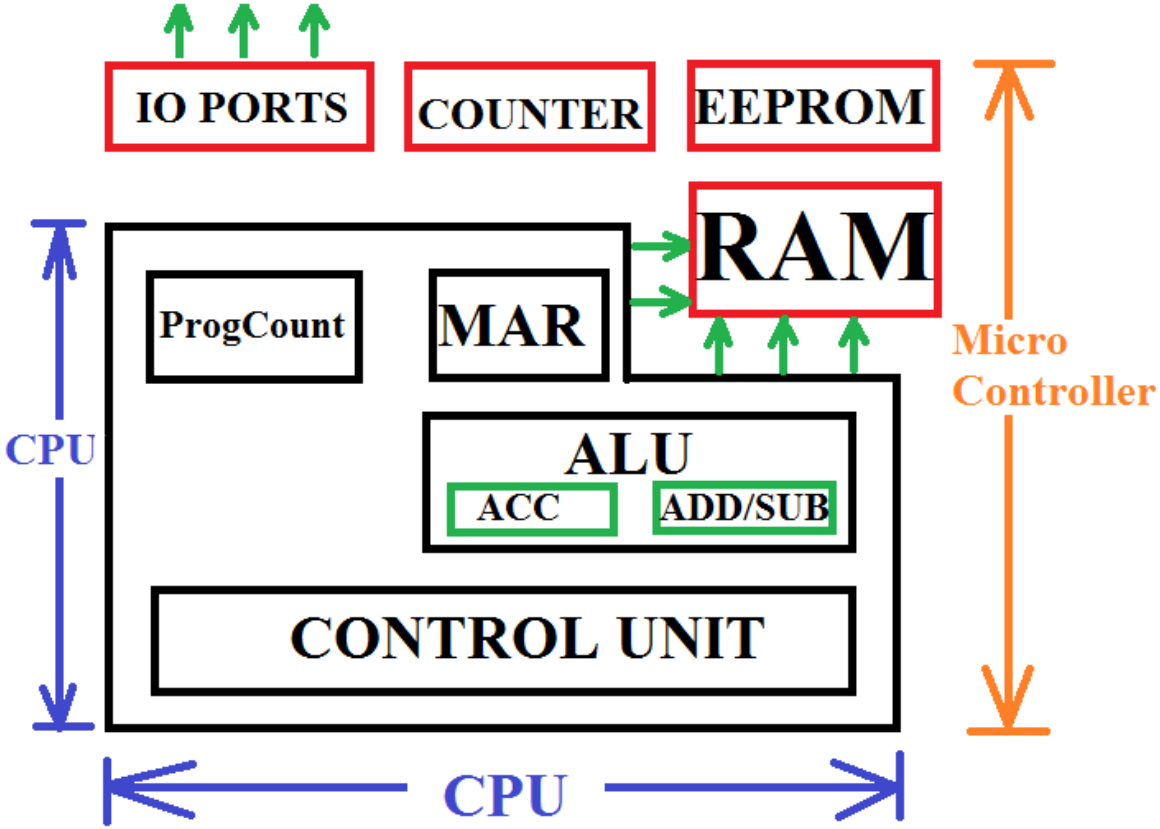


Fig. 3.2.1 Comparison of uC and uP chips

Notice that a uP will have address and data pin-outs while a uC will just have general purpose I/O pins P1,P2, P3,…

If you are tasked to design a heat control system to periodically read temperature, control heating, display current temperature, have a serial communication with computer for updating, the PCB layout for uP and uC will have the following requirements:

µP:

Microprocessor based system requires extra components (ICs):

- ✓ SRAM (for variables),
- ✓ Flash memory (for program),
- ✓ EEPROM (for constants),
- ✓ Timers (e.g. CTC),
- ✓ Input/Output (IO) e.g Serial IO (SIO chips) and Parallel IO (PIO chips).

µC:

The microcontroller will have all the components on **ONE CHIP** and hence reduced PCB area.

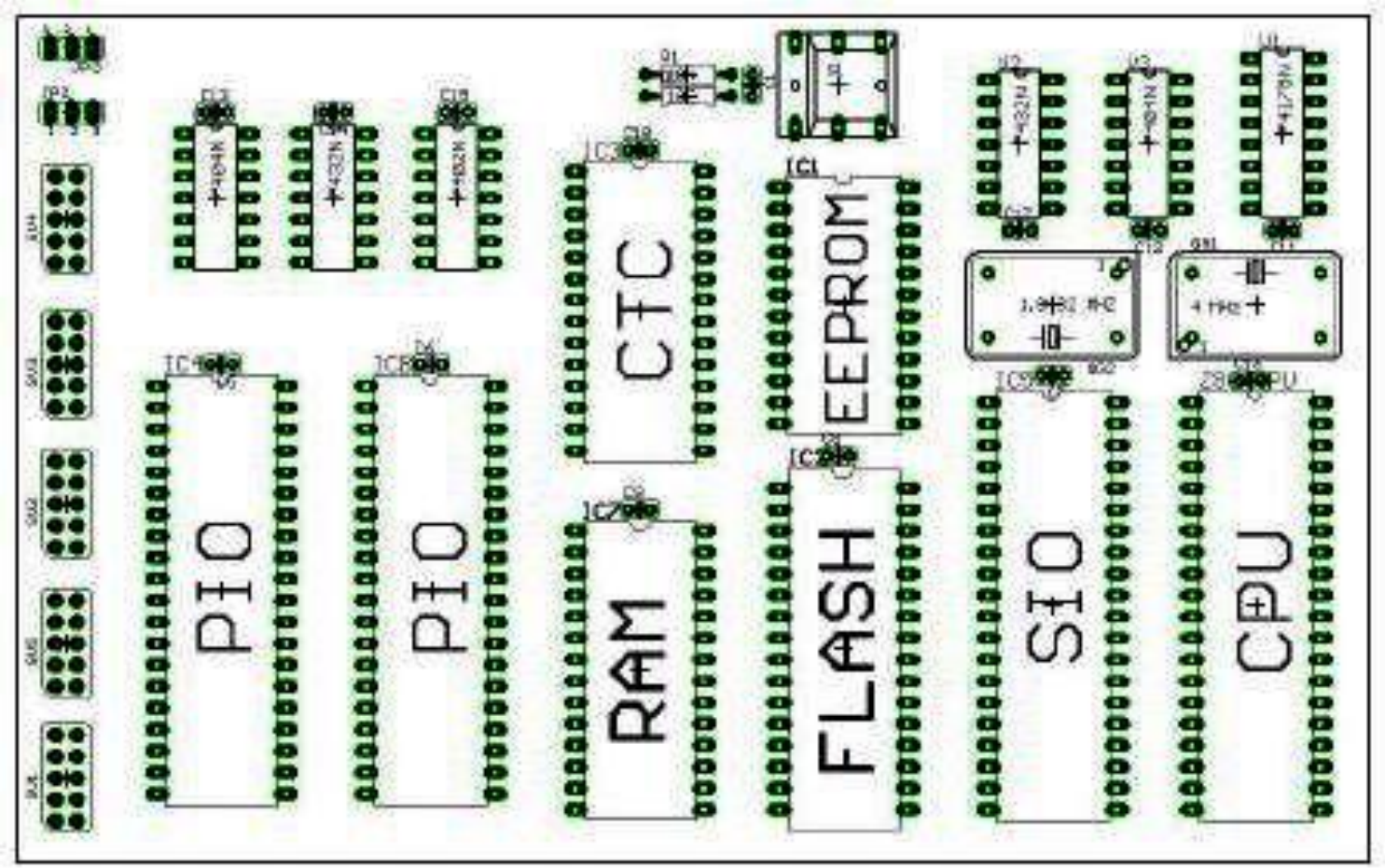


Fig. 3.2.2 Microprocessor Based System PCB

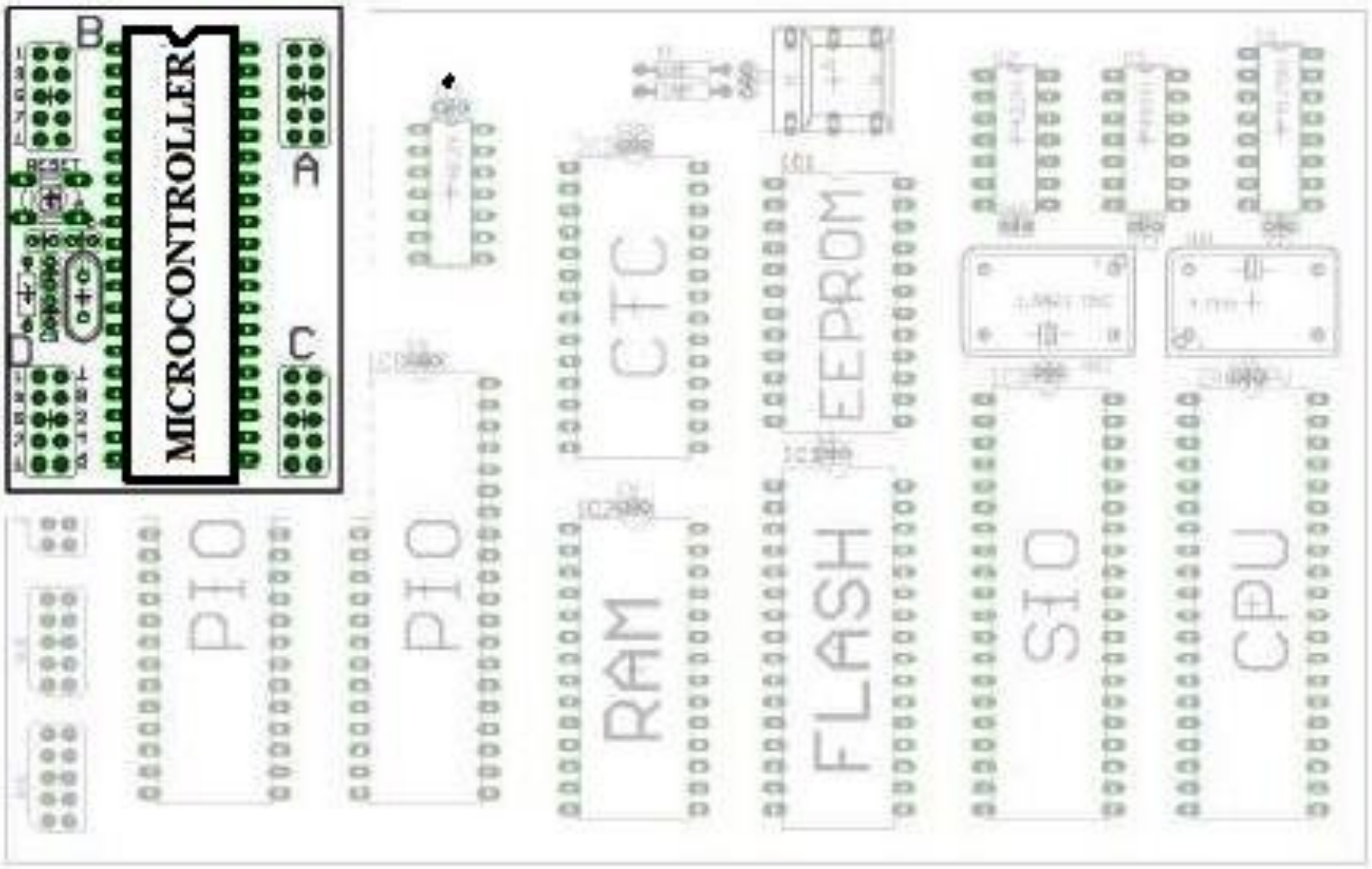


Fig. 3.2.3 Microcontroller PCB

Examples Microprocessors are:

- ✓ Intel's 8085, 8086, 8088, Pentium II, Duo Core, Core i7 etc
- ✓ Zilog's Z80, etc

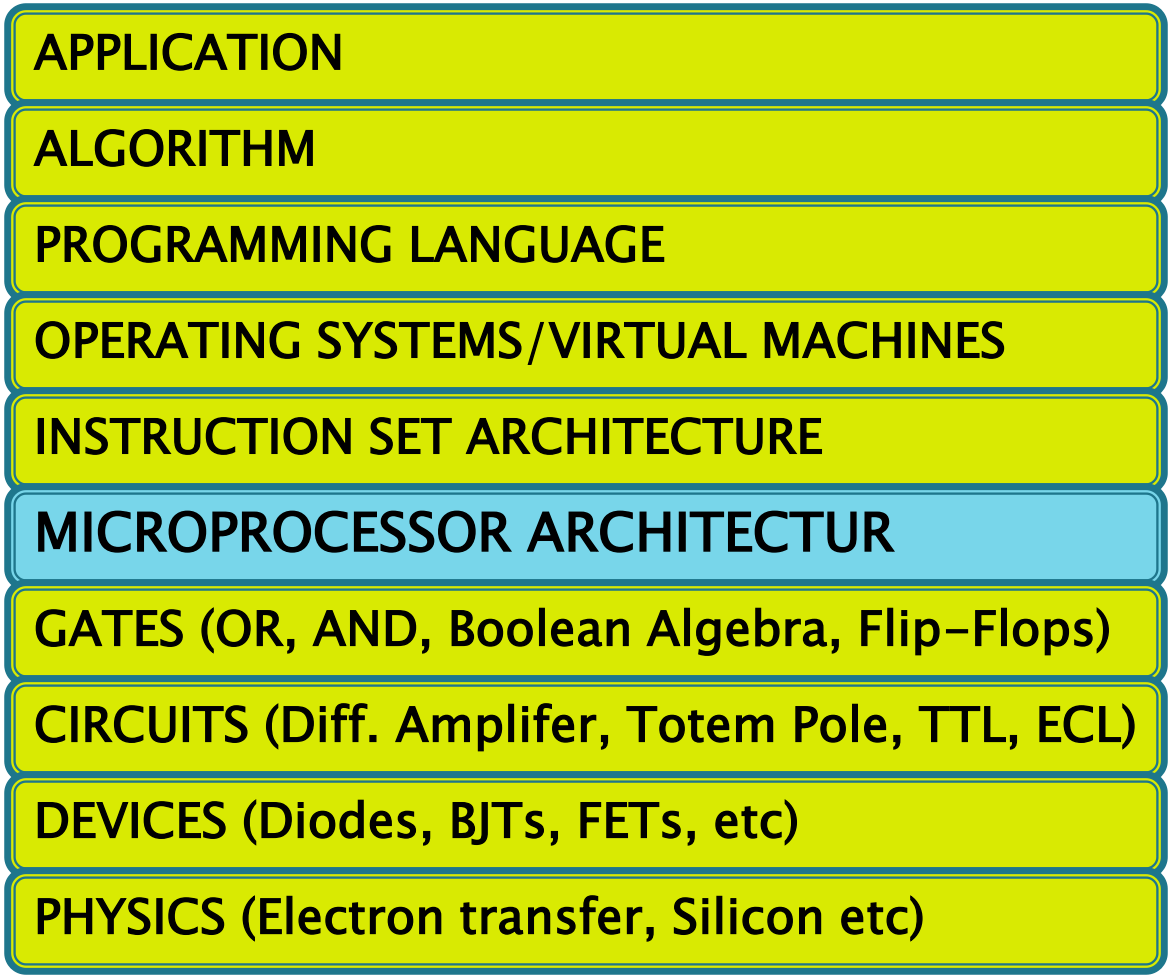
Differences being number of bits for data and address buses, clock speeds etc. e.g 8085 and Z80 where 8-bit µPs (4th YEAR).

Examples of Microcontrollers are:

8051, 8052, and their derivatives, Zilog's Z8, Motorola's 6811, Microchip's PIC, Atmel's AT family (ATMega, ATTiny) etc.

Students usually mistake the number 8085 (µP) for 8051 (µC) and the other way round.

What we are going to study is general architecture (structure) of a CPU-based system (can be extended to both uP and uC).



What is inside a microprocessor (CPU)?

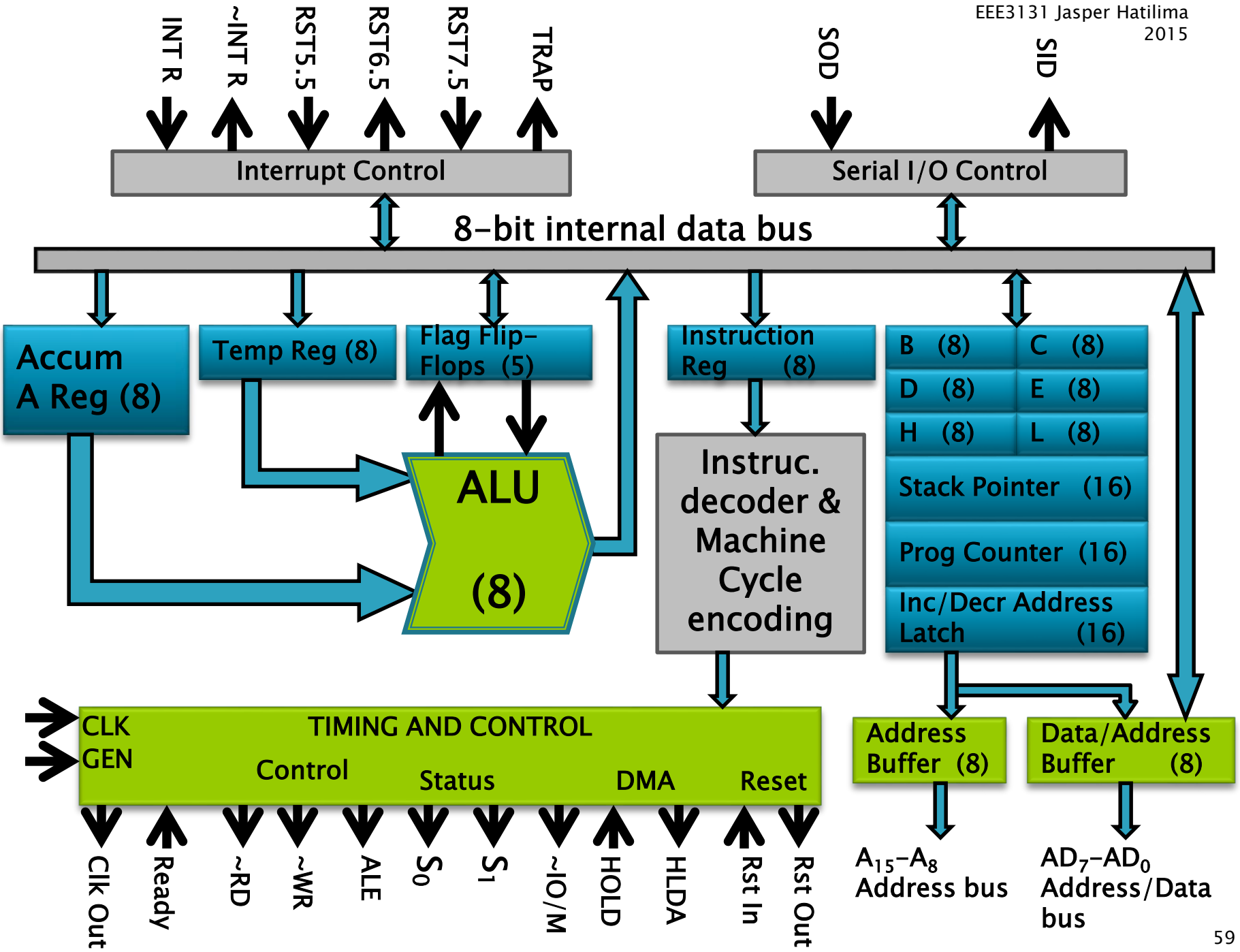
We will specifically look at the architecture of Intel's 8085.

Concepts learnt here can easily be extended to other architectures like the 8086, Z80 or the current ones and to µC too.

8085

- ✓ **8-bit** CPU
- ✓ **3-6MHz**
- ✓ **16-bit** address
- ✓ **8-bit** accumulator
- ✓ Six General Purpose registers each **8-bit**: B, C, D, E, H, L.
- ✓ 40 pins, X_1 and X_2 are for crystal oscillator.

The 8085 architecture is show in Figure 3.2.4 below:


INT R
~INT R
RST5.5
RST6.5
RST7.5
TRAP
SOD
SID
Interrupt Control
Serial I/O Control
8-bit internal data bus
Accum
A Reg (8)
Temp Reg (8)
Flag Flip-
Flops (5)
Instruction
Reg (8)
B (8)
C (8)
D (8)
E (8)
H (8)
L (8)
Stack Pointer (16)
Prog Counter (16)
Inc/Decr Address
Latch (16)
ALU
(8)
Instruc.
decoder &
Machine
Cycle
encoding
CLK
GEN
TIMING AND CONTROL
Control
Status
DMA
Reset
Address
Buffer (8)
Data/Address
Buffer (8)
Clk Out
Ready
~RD
~WR
ALE
S0
S1
~IO/M
HOLD
HLDA
Rst In
Rst Out
A15-A8
Address bus
AD7-AD0
Address/Data
bus

Description of the 8085 components

1. **Control Unit**: Gets the decoded instruction and uses it to generate control signals that will close and open connections between blocks. Inside it has a array of gates that take an input instruction e.g. 1011 0001 and then produce a certain combination of output signals e.g. READ=0, WRITE=1, HOLD = 0, etc.
2. **Arithmetic and Logic Unit** (**ALU**): Performs the actual numeric and logic operations. Has adder-subtractor, OR, AND, etc inside. Accumulator is part of ALU.
3. **Accumulator**: Is an 8-bit register that is part of ALU. Stores data used in the arithmetic operations. It also stores results of the operation. ***It is also called Register A***.
4. **Instruction Register/Decoder**: Temporarily store the current instruction of the program received from the memory. The IR then passes the instruction to the decoder for interpretation. The decoder then passes it to the control unit.

5. **Memory Address Register**: This is an address buffer. It receives the address of the next instruction and then holds it before placing it on the Address Bus.
6. **Registers**: The 8085 has an accumulator register, a flag register, six general purpose registers (8-bits), two 16-bit registers: the Stack Pointer and the Program Counter. Below are the details about these registers:
   i. **General Purpose Registers**: Store 8-bit data and are identified as B, C, D, E, H and L. In order to perform 16-bit operations, they can be paired into BC, DE and HL.
   ii. **Flag Registers**: Organized as five flip-flops that are set/reset according to data conditions in accumulator. Since this is an -bit register, only five of the bits out of eight are used. The five flags represented by the flip-flops are **Zero** (Z), **Carry** (CY), **Sign** (S), **Parity** (P), and **Auxiliary Carry** (AC). Example is when the result of an operation is more than 8-bits, the CY flag will be set. This can useful when using Assembly instructions like JC (Jump on Carry).

iii. **Program Counter**: Contains the address (16 bits) of the instruction that will be executed in the next step. So this is a counter that runs from 0000 0000 0000 0000 to 1111 1111 1111 1111.

iv. **Stack Pointer**: The SP is a 16-bit register used as a memory pointer ->it points to the stack.
The STACK is a LIFO (Last In First Out) register (In the RAM) where **local variables** and **return addresses** are stored to allow simple nesting of function calls in a PROGRAM.
You **PUSH** items onto the stack and **POP** them off.

FROM OUTSIDE, THE IC HAS THE FOLLOWING PIN-OUT:

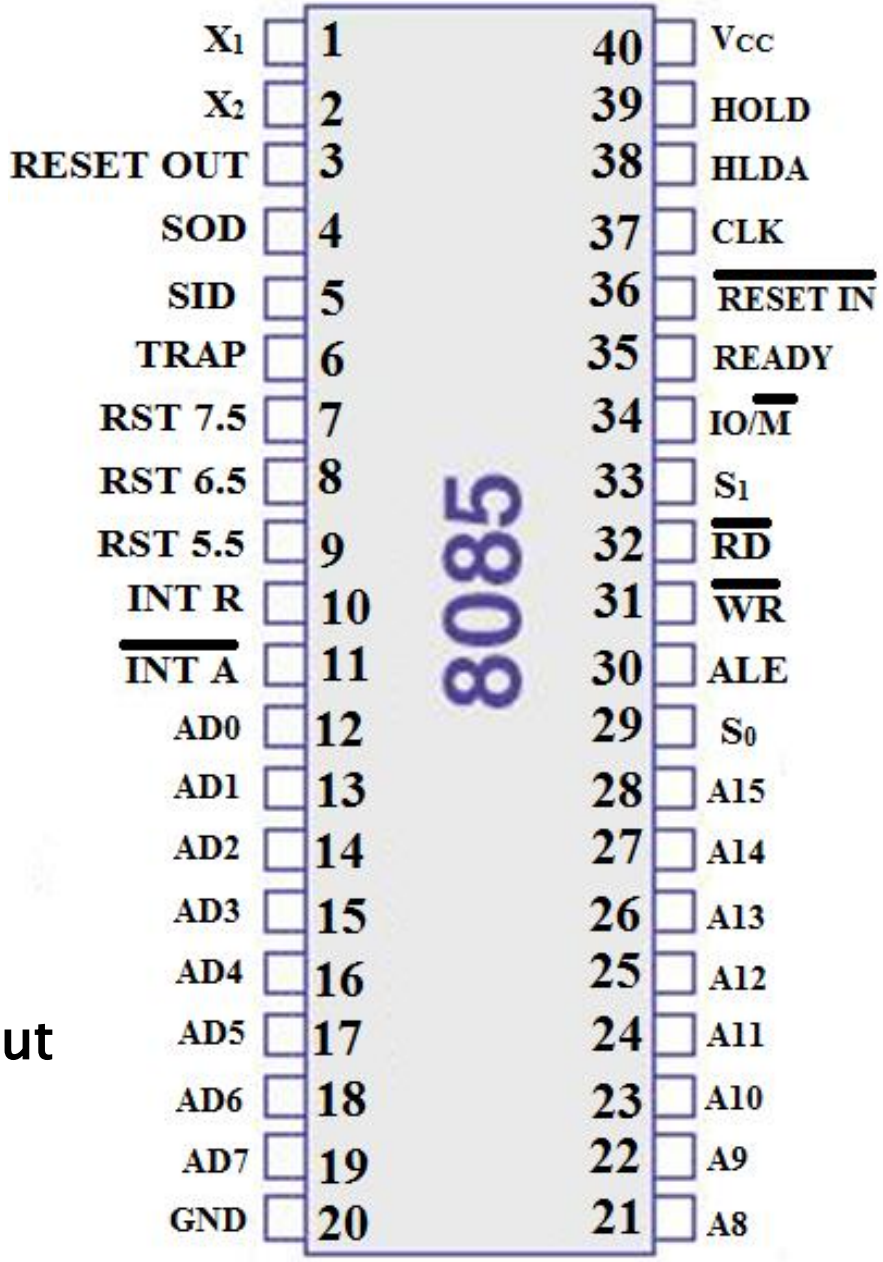


Fig. 3.2.5 DIP 8085 Pin Out

In the next section, we will look at how the microprocessor connects to the other sections of the system to make a functional microprocessor based computer system...

This takes us to the concept of a **Bus Organized Computer**

Bus Organised Computer

A bus is a collection of wires that transmit binary numbers, one bit per wire.

The bus structure for the 8085 are as follows:

1. Address Bus

✓ The address has a width of **16 bits** therefore the address bus has 16 wires.

✓ Able to address 2^{16} = **65,536 memory locations** (1 byte each) from 0000 0000 0000 0000 to 1111 1111 1111 1111.

✓ Address bus is **unidirectional** i.e. addresses are only sent from µP to memory and never the other way.

2. Data Bus

✓ Data bus has a width of **8 bits** = 8 wires. Carries the bits representing the **DATA OR INSTRUCTIONS** between µP and external components (**BI-DIRECTIONAL**)

✓ In terms of data, it can carry 2^8 = **256 different combinations of data or instructions**.

✓ In terms of data values, the **largest number that it can carry is 1111 1111** (Decimal 255). Larger numbers have to be represented by several bytes thus slowing the µP.

✓ In terms of instructions, the data bus limits the number of **possible instructions for this architecture to 256**.

3. Control Bus

Using the received instruction, the timing and control unit generates the necessary control signals that direct the various components.

✓ These control signals, e.g. READ/WRITE, HOLD, RESET etc, are then carried from the control unit using the control bus.

✓ Control bus has a width of **8-bits** = 8 wires (**UNI-DIRECTIONAL**).

✓ It can carry 2^8 = **256 different combinations of control signals**.

Below is a diagram showing a bus organized computer.

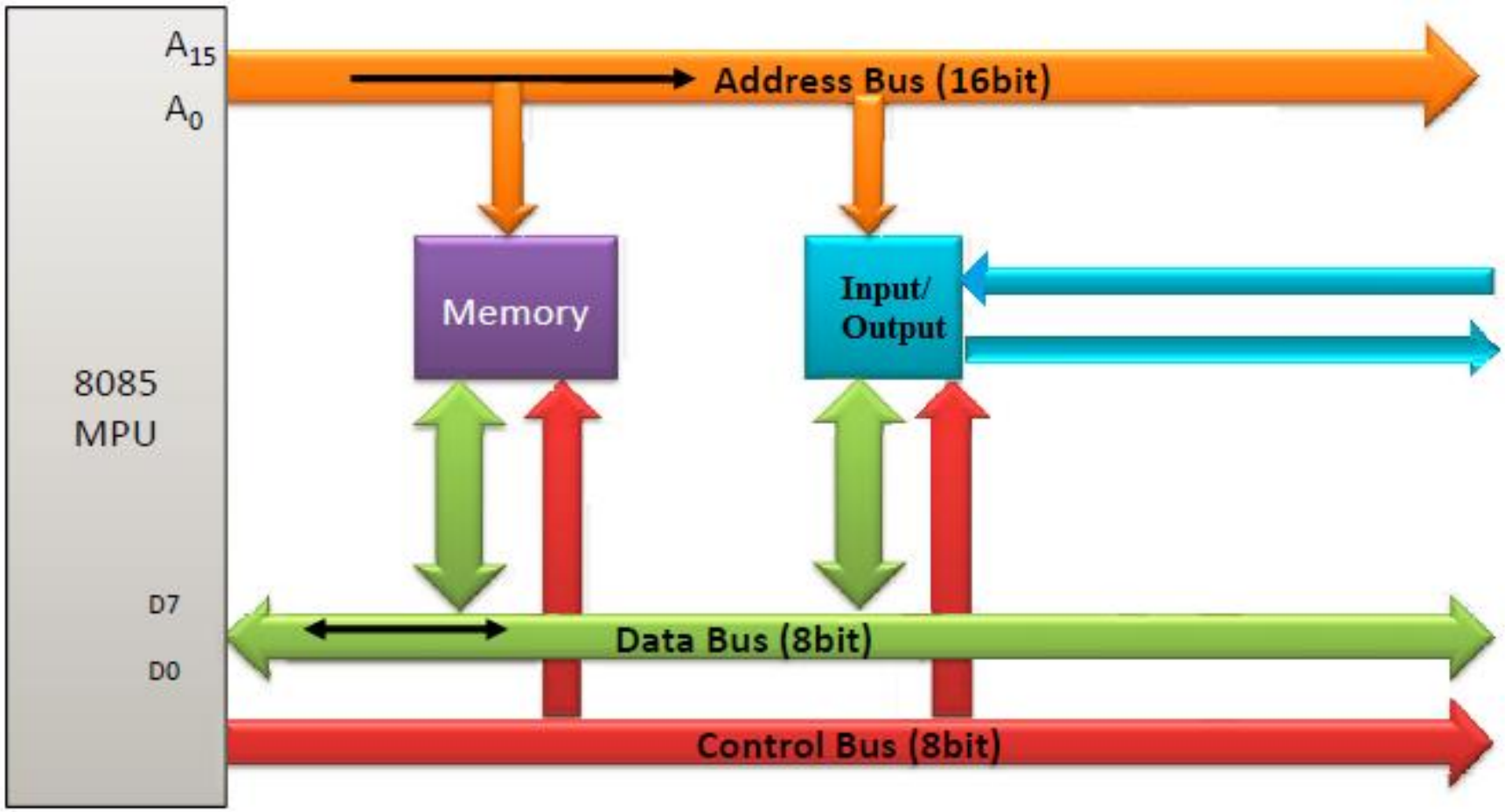


Fig. 3.2.6 Bus Organized Computer

INTRODUCTION TO COMPUTER PROGRAMMING

For easier human computer interaction and for flexibility on how the computer will be used, users should have a quick and standard way of

- ✓ changing the operations of the computer,
- ✓ entering new data,
- ✓ reading data from the system etc.

This brings us to the concept of computer programming and microcontroller programming.

- ❑ Programming can be done in **MACHINE LANGUAGE** (binary) but it would be tedious. This is a low level language.

- ❑ Another low level language is **ASSEMBLY LANGUAGE**.
- ❑ **High level languages** are human readable (English) include C, C++, Java, Python etc

ASSEMBLY LANGUAGE

Recall a few pages back when we studied a simple computer that instructions are loaded in the RAM as binary symbols.

Some of the instructions we would like to run are register loading instructions, addition of some register contents etc

As we showed earlier, these actions are represented by some standard short words like ADD for addition, SUB for subtraction, LDA for loading into A register etc.

These **short-form** words are called **MNEMONICS** and represent the action or operation that needs to be performed.

The above is specifically true for a language that is close to machine language where the Mnemonics have a direct binary representation... **ASSEMBLY LANGUAGE.**

✓ Assembly language is a low level language.

✓ Requires a utility program called an **ASSEMBLER** to convert the assembly language into executable machine code.

✓ There are several assembler programs:

- Microsoft Assembler (MASM)
- Borland Turbo Assembler (TASM)
- GNU Assembler (GAS)

The microprocessor simply executes instructions that were entered in form of **a program that resides** in memory (binary).

The microprocessor does this in what is called **fetch-decode-execute** cycle.

- ❑ Fetch the instruction from memory
- ❑ Decoding or 'identifying' the instruction
- ❑ Executing the instruction.

✓ As earlier stated, an assembler software translates the assembly language code into machine code (1's and 0's).

✓ Writing the assembly code (saved as **file.asm**), is done in a text editor like notepad, EMACs etc.

✓ Some special programming environments have been developed where the text editor and the assembler are integrated.

✓ These are called **Integrated Development Environments** (IDE).

✓ In assembly language, the instruction consists of two parts:

- ❑ Operation code (OPCODE) represented by a Mnemonic.
- ❑ Data/Address on which operation is to be performed (OPERAND). This is usually in HEXADECIMAL.

To make the code readable to the next user, comments are inserted using the semi-colon at their beginning. Comments are NOT EXECUTED.

An instruction in assembly language programming may also have a **LABEL.**

This is a character string used to label a certain potion of the code.

Between label and operand, a full colon is used,

A semicolon is used before a comment.

The overall instruction format is therefore as follows:

Label	Opecode (Mnemonic)	Operand (s)	Comment
Start :	MVI	C, 46h	; move value 70 into Reg C
Again :	DCR	C	; Decrement C
	JNZ	Again	; Continues to count down

8085 ASSEMBLY LANGUAGE MNEMONICS

1. Data Transfer Operations

MOV B, A ; Move contents of Reg A to Reg B
LDA B ; Load Accumulator with contents in Reg B
STA B ; Store accumulator contents in Reg B
MVI B, 24h ; Move Immediate to Reg B the value 24 (hex)

2. Arithmetic Operations

ADD B ; Add Reg B contents to Reg A, i.e. A = A+B
SUB B ; Subtract Reg B contents from Reg A, i.e. A = A-B
INC R ; Increment contents of Reg R by 1 i.e. R=R+1
DCR R ; Decrement contents of R by 1 i.e. R=R-1

3. Logical Operations

CMA ; Complement the accumulator bit-by-bit

ANA B ; means 'AND' accumulator with indicated register i.e. bit-by-bit AND operation of Reg B and Accumulator.

ORA B ; means 'OR' accumulator with indicated register i.e. bit-by-bit OR operation of Reg B and Accumulator.

XRA B ; means 'XOR' accumulator with indicated register i.e. bit-by-bit XOR operation of Reg B and Accumulator.

ANI byte ; means 'AND Immediate'. AND operation of accumulator with the byte that immediately follows the instruction. E.g. If A = 0101 1110, ANI C7h is and operation of 0101 1110 with 1100 0111 which is 0100 0110.

ORI byte, XRI ; OR immediate and XOR immediate have similar structure as ANI above.

4. Jump Operations(Conditional Jumps)

These change the program sequence.

JMP 234h ; Mean jump. Once execution reaches this line, the program counter is set to address 234h, i.e. execution jumps to 234h (this address value is just an example)

JM 2006h ; Jump if minus=> if the result of some execution has triggered the sign flag, execution jumps to memory 2006h.

JZ 234h; Jump if zero. Execution jumps to the specified memory location if the zero flag is set (accumulator contents become zero after an operation)

JNZ 234h ; Jump if not zero. Execution jumps to the specified memory location if the zero flag is not set (ACCUM not zero)

5. Other Operations

NOP; Means No Operation. No register changes occur during this instruction. It is used to waste time…delay. The exact delay of one NOP depends on the processor.

HLT; Means Halt. This instruction stops the execution.

IN 02H; Means Input. Transfers data from specified input port to the accumulator.

OUT 04H; Means Output. Transfers data from accumulator to the specified port.

RAR; Means Rotate Accumulator Left e.g. if A = 1011 0110, then RAL will produce A = 0101 1011

RAL; Means Rotate Accumulator Right.

Example of simple assembly language program:

```
>> MVI A, 24h  ; Load Accum with value 24 (hex)
>> MVI B, 56h  ; Load B register with value 56 (hex)
>> ADD B       ; Add B register values to Accum A = A+B
>> OUT 01h     ; Display the Accum contents on port
                 addressed 01 (hex)
>> HLT         ; Stop execution
```

The result of the operation should be 7Ah.

Notice the various ways the address of the data to be operated on is specified.

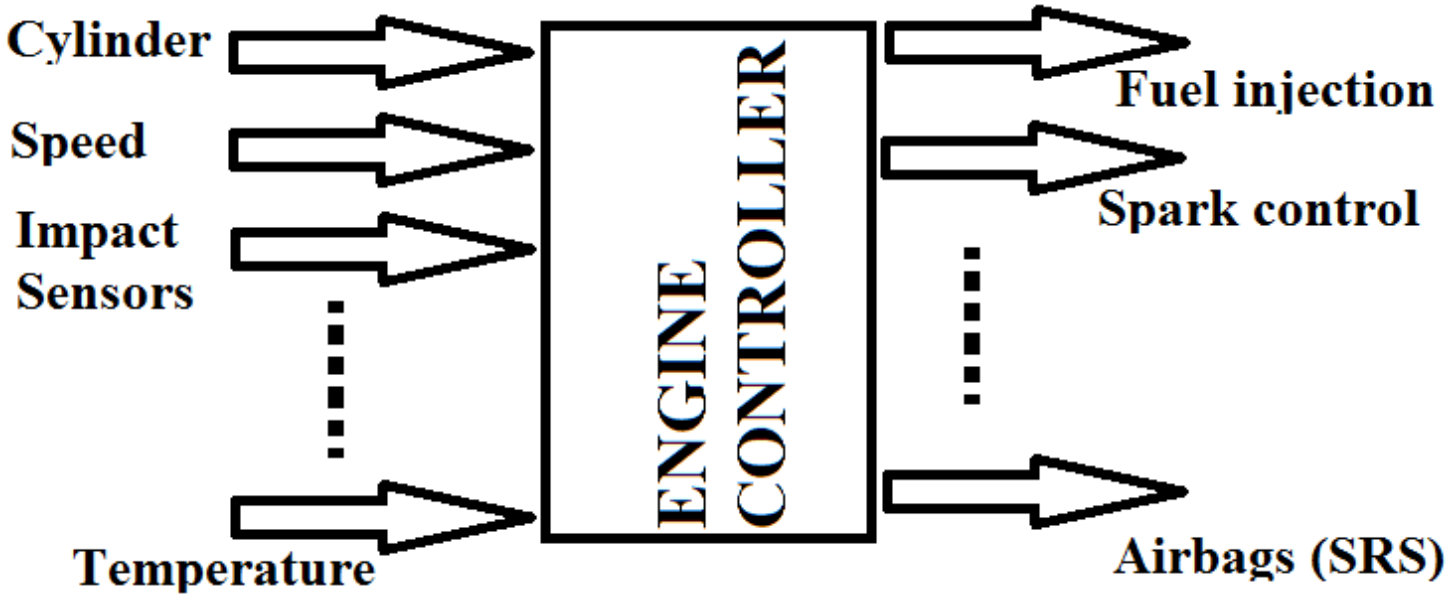
CLOSING ON MICROPROCESSORS AND MICROCONTROLLERS

So now you understand microprocessor architecture and basics of assembly language to program controllers.

You also understand the difference between a microprocessor and a microcontroller.

Since a microcontroller has additional peripheral devices like A/D and D/A converters, they are used in special purpose systems.

Such a special purpose computer system is called an **EMBEDDED SYSTEM** – It is not used for general purposes *(Hoping this course will be introduced in the future)*.
An example is the auto-processing unit in **automobiles**.



By 1990:

Toyota was using its own µC. It refered to its µC based system as *Engine Control Unit* (ECU).

Audi used Motorola's 6802 for its controls.

Ford collaborated with Intel to have an 8061 used for its cars.

The process of making customized microcontrollers brings us to the next and last topic... **PLD**s.

PART 3.3
PLDs: PLA, PAL and FPGAs

Any definitions, descriptions, ideas etc from the class?

PLDs:

Programmable Logic Devices (PLDs) are general purpose microchips that are used to implement logic circuits.

A PLD can be viewed as a 'black box' that is made up of **logic gates** and **programmable switches**.

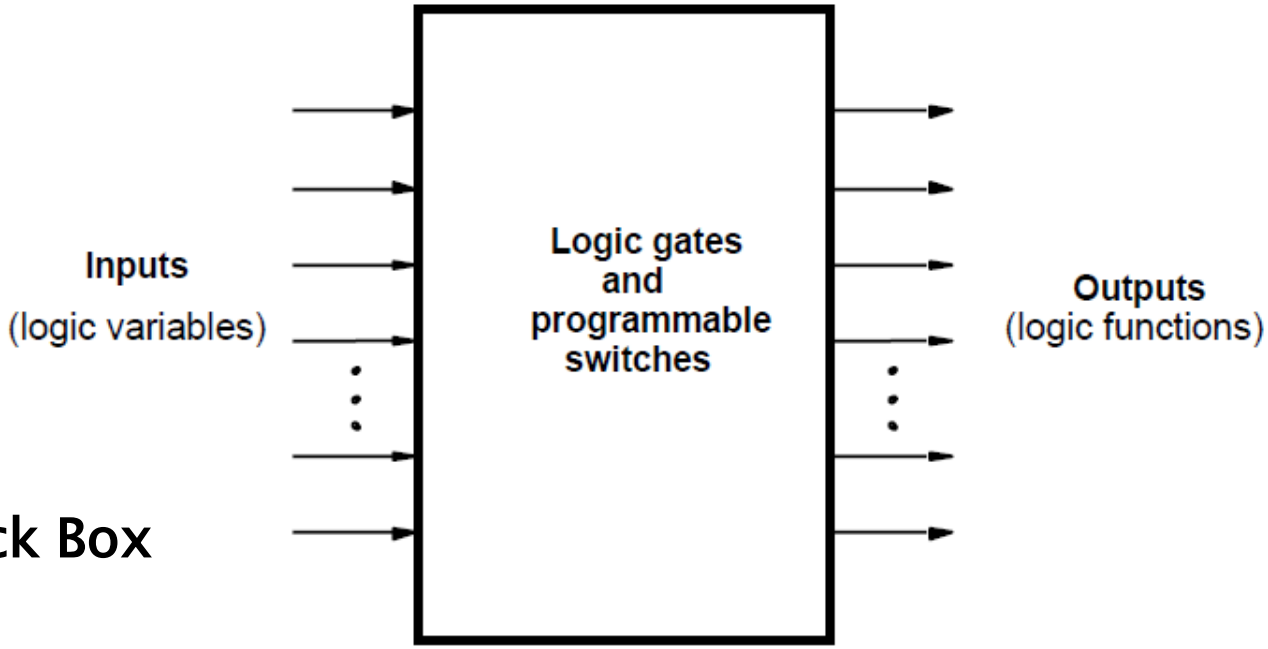


Fig. 3.3.1 PLD Black Box

CLASSIFICATION OF PLDs

PLDs are generally classified into three categories:

- Simple PLDs (SPLDs)
  - Programmable Logic Array (PLA)
  - Programmable Array Logic (PAL)
- Complex PLDs (CPLDs)
- Field Programmable Gate Arrays (FPGAs)

Summarizing EEE 3131

✓ As we have seen, we can implement logic designs from transistor level using TTL, ECL, CMOS etc and **obtain logic gates**.

✓ We can then use these **logic gates to implement sub-systems** like Flip-flops, counters, registers, control units, A/D, D/A etc.

✓ We can then integrate these components to have a **fully functional digital computer system**. We looked at **µP** and **µC**.

Future courses that will depend on the knowledge gathered here:

EEE 4131 - Computer Engineering: High level languages, Classification of processors, Multiprogramming, Multiprocessing etc

Thank You All