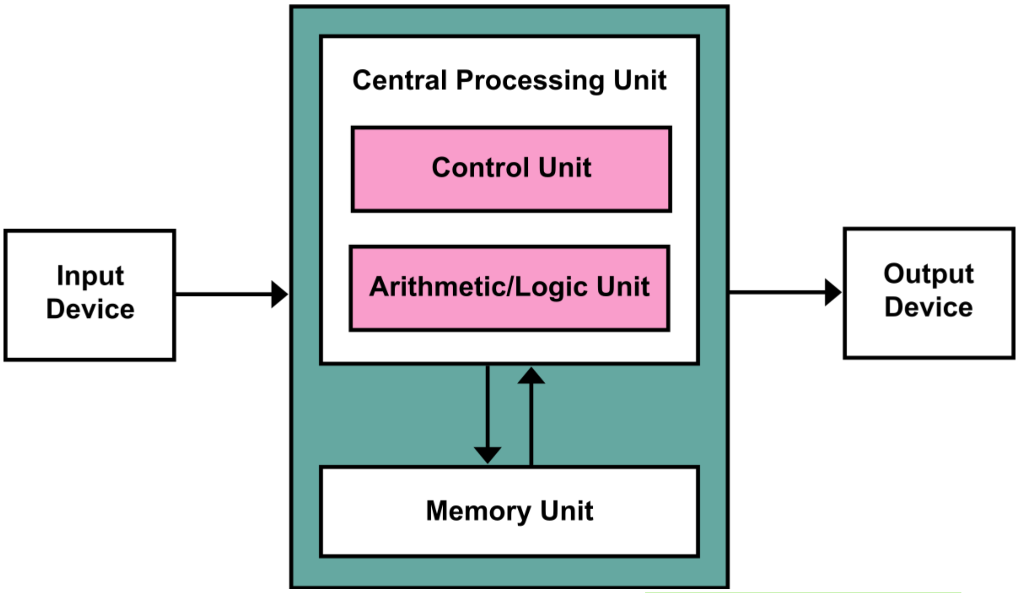
Functional units of computer:



A computer is not a single unit but it consists of many functional units(intended to perform jobs) such as Input unit, Central Processing Unit(ALU and Control Unit), Storage (Memory) Unit and Output Unit.

1. Input Unit: Its aim is to supply data (Alphanumeric, image , audio, video, etc.) to the computer for processing. The Input devices are keyboard, mouse, scanner, mic, camera, etc

2. Central Processing Unit (CPU): It is the brain of the computer and consists of three components

Arithmetic Logic Unit(ALU): As the name implies it performs all calculations and comparison operations.

Control Unit(CU): It controls overall functions of a computer

Registers: It stores the intermediate results temporarily.

3. Storage Unit(Memory Unit): A computer has huge storage capacity. It is used to store data and instructions before starts the processing. Secondly it stores the intermediate results and thirdly it stores information(processed data), that is the final results before send to the output unit(Visual Display Unit, Printer, etc)

Two Types of storage unit

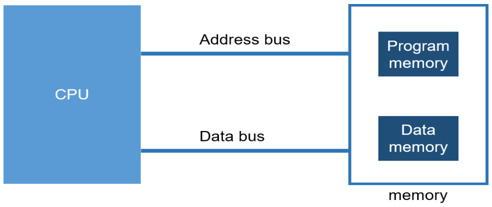
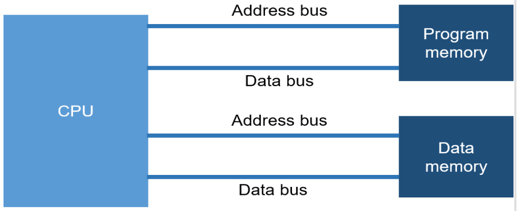
(a) Primary Storage alias Main Memory: It is further be classified into Two- Random Access Memory(RAM) and Read Only Memory(ROM). The one and only memory that the CPU can directly access is the main memory at a very high speed. It is expensive hence storage capacity is less. RAM is volatile (when the power is switched off the content will be erased) in nature but ROM is non volatile(It is permanent)

(b) Secondary Storage alias Auxiliary Memory: Because of limited storage capacity of primary memory its need arises. When a user saves a file, it will be stored in this memory hence it is permanent in nature and its capacity is huge. eg: Hard Disc Drive(HDD), Compact Disc(CD), DVD, Pen Drive, Blu Ray Disc etc.

4. Output Unit: After processing the data we will get information as result, that will be given to the end user through the output unit in a human readable form. Normally monitor and printer are used.

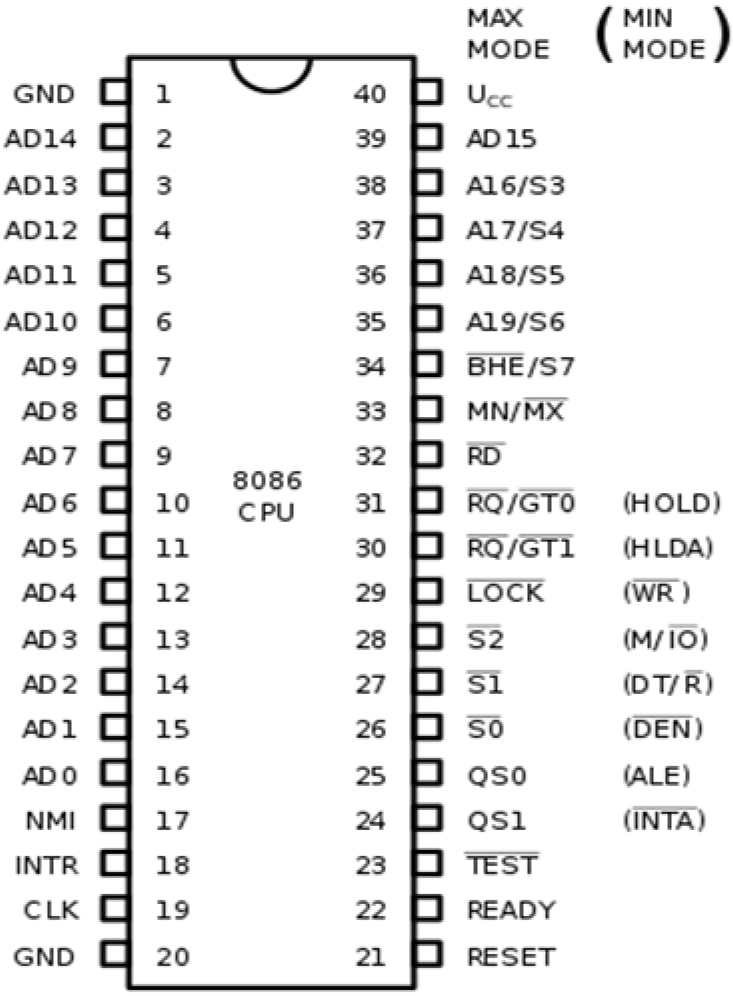
Difference between von neumann and harvard architecture

Key important difference between von neumann and harvard architecture are given below

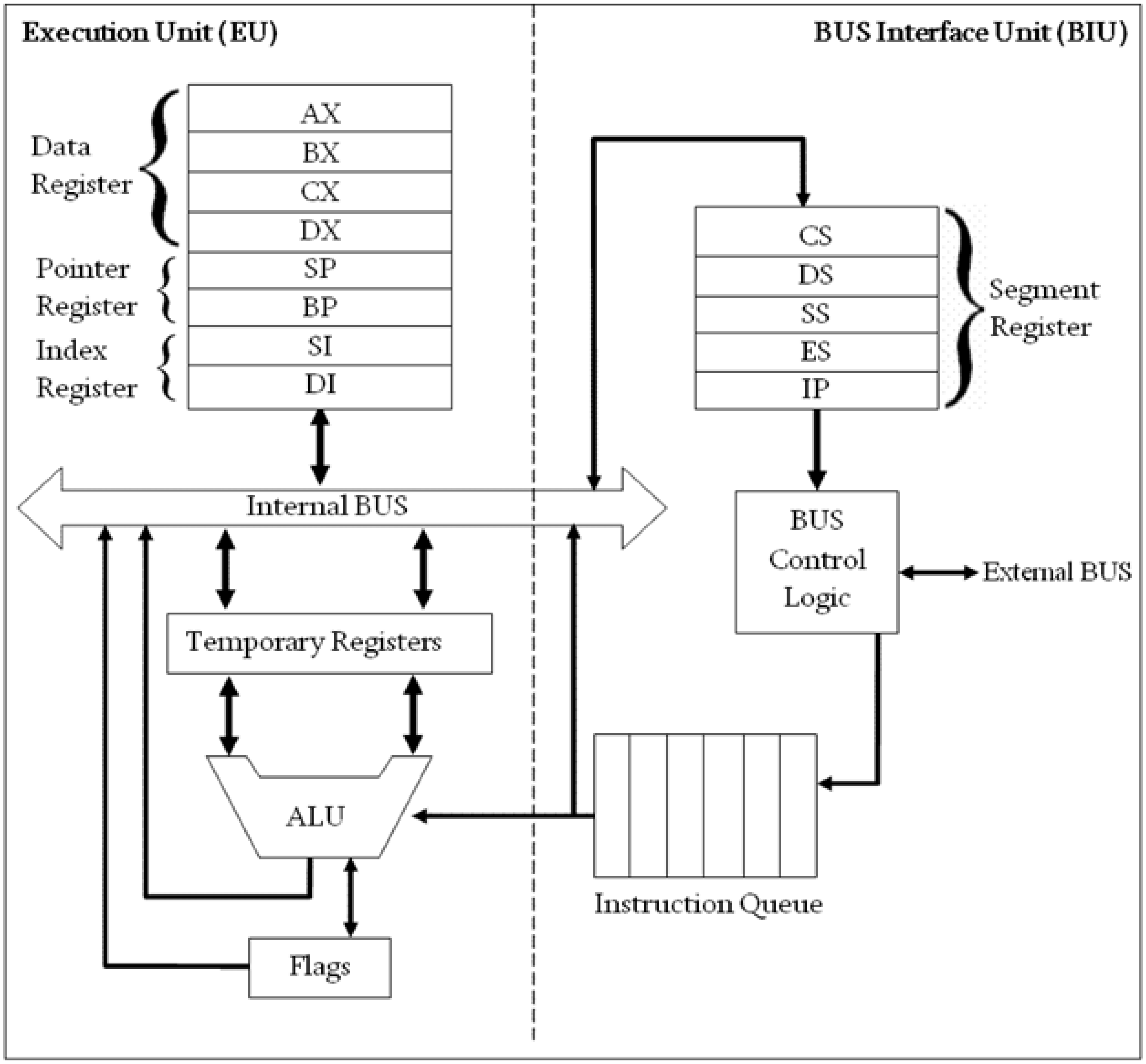
Parameter	**Von neumann architecture**	**Harvard architecture**
1)Defination	The architecture which uses common bus to access data memory and program memory is called as von neumann architecture	The architecture which uses separate address bus and data bus to access data memory and program memory respectively is called as harvard architecture
2)Block diagram	 CPU Address bus Data bus Program memory Data memory memory Von neumann architecture	 CPU Address bus Data bus Address bus Data bus Program memory Data memory Harvard architecture
3)Storage	The von neumann architecture uses single memory space for both program memory and data memory	The harvard architecture uses seperate program and data memory space
4)Access	In von neumann architecture we can not access program and data memory simultaneously	In harvard architecture we can access program and data memory simultaneously because it has separate program memory and data memory

5)Architecture	The von neumann architecture uses CISC architecture	The harvard architecture uses RISC architecture
6)Execution of instruction	Execution of instruction takes more machine cycle	Execution of instruction takes less machine cycle

Introduction to microprocessor:



Functional units of 8086



What is an Assembler?

We know that assembly language is a less complex and programmer-friendly language used to program the processors. In assembly language programming, the instructions are specified in the form of mnemonics rather in the form of machine code i.e., 0 and 1. But the microprocessor or microcontrollers are specifically designed in a way that they can only understand machine language.

Thus assembler is used to convert assembly language into machine code so that it can be understood and executed by the processor. Therefore, to control the generation of machine codes from the assembly language, assembler directives are used. However, machine codes are only generated for the program that must be provided to the processor and not for assembler directives because they do not belong to the actual program.

8086- Assembler directives

Introduction:

Assembler directives are the directions to the assembler which indicate how an operand or section of the program is to be processed. These are also called pseudo operations which are not executable by the microprocessor. The following section explains the basic assembler directives for 8086.

ASSEMBLER DIRECTIVES:

The various directives are explained below.

1. ASSUME : The ASSUME directive is used to inform the assembler the name of the logical segment it should use for a specified segment.

Ex: ASSUME DS: DATA tells the assembler that for any program instruction which refers to the data segment ,it should use the logical segment called DATA.

2.DB -Define byte. It is used to declare a byte variable or set aside one or more storage locations of type byte in memory.

For example, CURRENT_VALUE DB 36H tells the assembler to reserve 1 byte of memory for a variable named CURRENT_ VALUE and to put the value 36 H in that memory location when the program is loaded into RAM .

3. DW -Define word. It tells the assembler to define a variable of type word or to reserve storage locations of type word in memory.

4. DD(define double word) :This directive is used to declare a variable of type double word or restore memory locations which can be accessed as type double word.

5.DQ (define quadword) :This directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory .

6.DT (define ten bytes):It is used to inform the assembler to define a variable which is 10 bytes in length or to reserve 10 bytes of storage in memory.

7. EQU –Equate It is used to give a name to some value or symbol. Every time the assembler finds the given name in the program, it will replace the name with the value or symbol we have equated with that name

8.ORG -Originate : The ORG statement changes the starting offset address of the data.

It allows to set the location counter to a desired value at any point in the program.For example the statement ORG 3000H tells the assembler to set the location counter to 3000H.

9 .PROC- Procedure: It is used to identify the start of a procedure. Or subroutine.

10. END- End program .This directive indicates the assembler that this is the end of the program module.The assembler ignores any statements after an END directive.

11. ENDP- End procedure: It indicates the end of the procedure (subroutine) to the assembler.

12.ENDS-End Segment: This directive is used with the name of the segment to indicate the end of that logical segment.

Ex: CODE SEGMENT : Start of logical segment containing code

CODE ENDS : End of the segment named CODE.

Module 2
Overview of Computer Architecture & Organization

https://www.youtube.com/watch?v=jTa0w-MxFJE

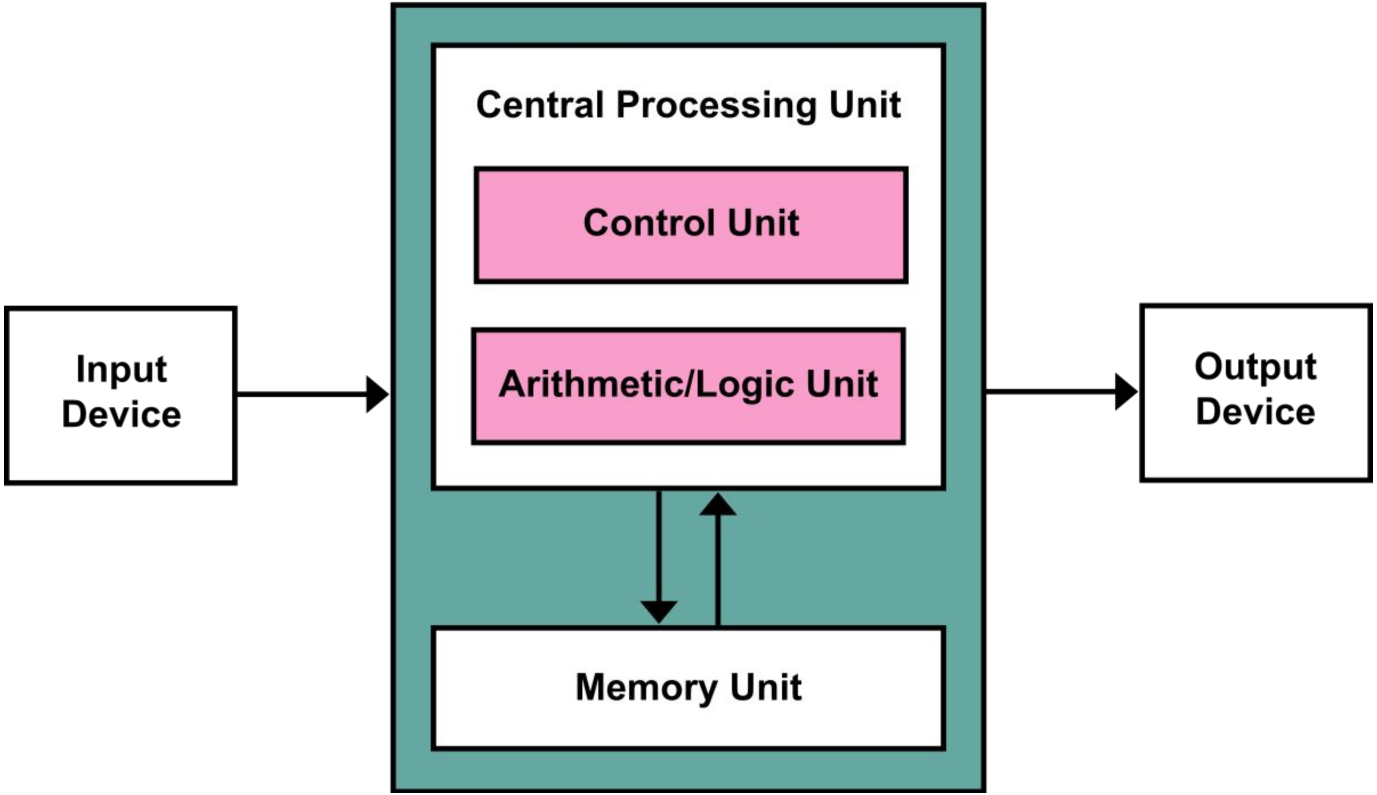
Introduction

- A computer is a combination of **hardware and software** resources which integrate together and provides various functionalities to the user.
- Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc.
- While software is the set of programs or instructions that are required by the hardware resources to function properly.

Introduction

- **Computer architecture** describes those attributes of the system that are visible to the user.
- **Computer organization** deals with implementation of these features and is mostly unknown to the user.

Functional units of computer



Functional units of computer

The components of computer are CPU, memory and I/O devices.These units are connected to each other with the help of buses.

1. **Memory:** It is used to store program and data.
2. **I/O devices:** It is used to accept input and produce desired output.

Functional units of computer

3. **CPU:** It consist of ALU and control unit.
    - ALU: It performs arithmetic operations.
    - Control unit: It stores instructions into the register and gives signal in proper sequence to execute a instruction.

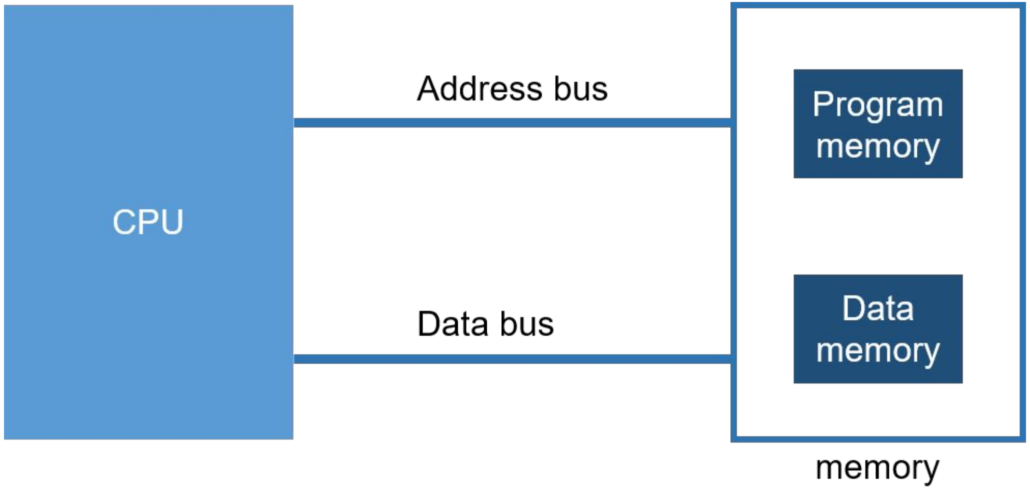
The functions of these units are data processing, data storage , data movement and control.

Von Neumann & Harvard Architecture

- There are basically two types of digital computer architectures.
- The first one is called Von Neumann architecture and later Harvard architecture was adopted for designing digital computers.

Von Neumann Architecture

The architecture which uses common bus to access data memory and program memory is called as von neumann architecture



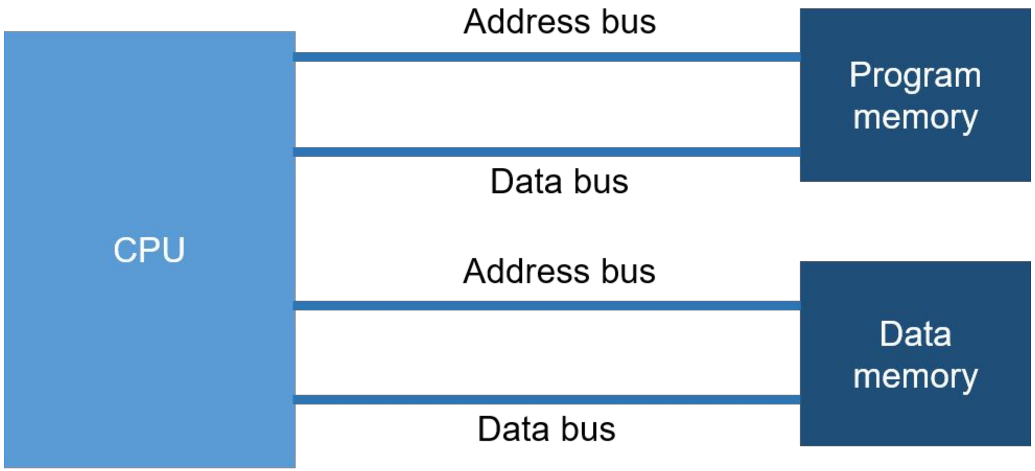
Von neumann architecture

Von Neumann Architecture

- One shared memory for instructions (program) and data with one data bus and one address bus between processor and memory.
- Instructions and data have to be fetched in sequential order (known as the Von Neuman Bottleneck), limiting the operation bandwidth. Its design is simpler than that of the Harvard architecture.
- Most of the general purpose microprocessors such as motorola 6800, intel 8086 use this architecture.

Harvard Architecture

- Harvard architecture uses separate buses for instructions and data.

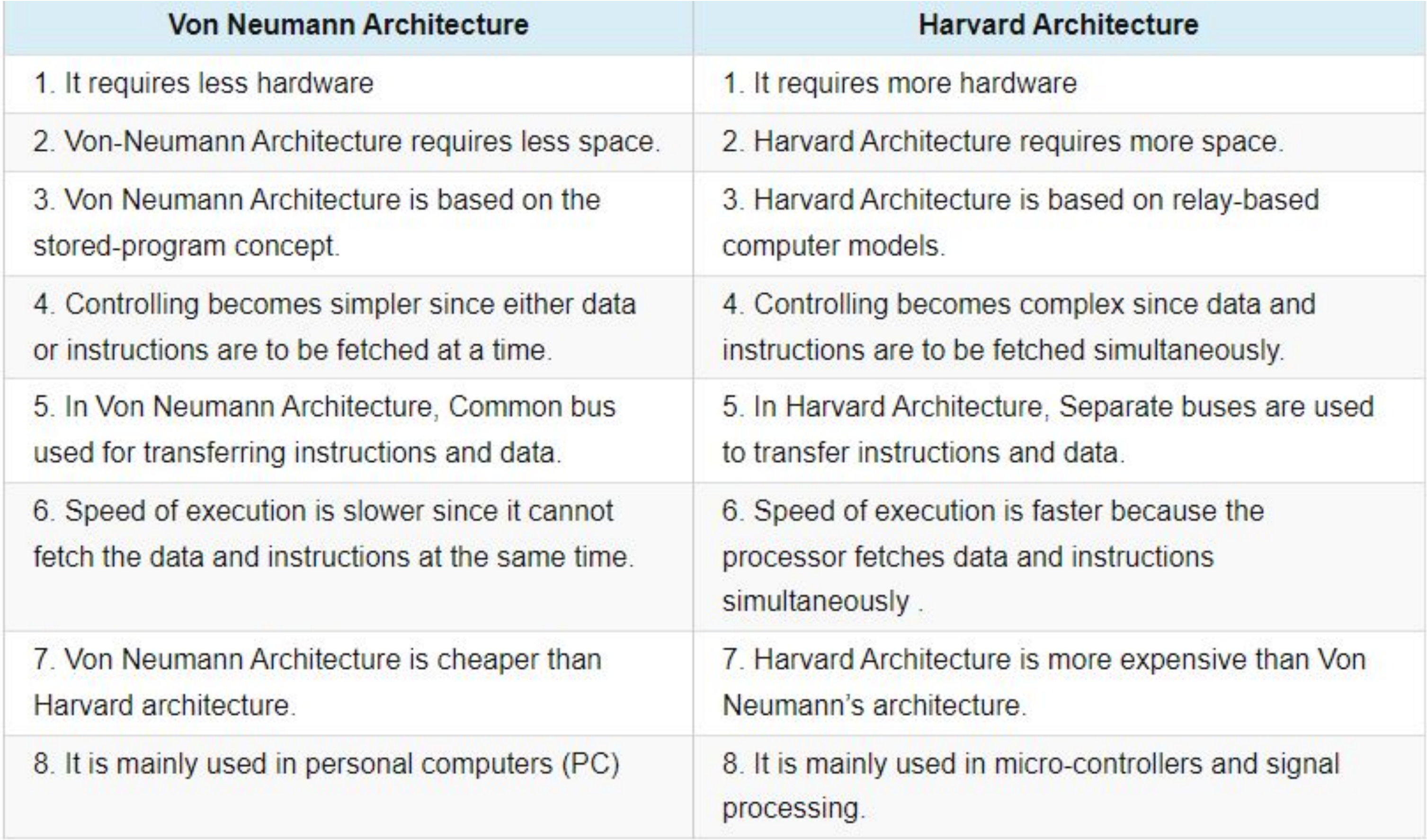


Harvard architecture

Harvard Architecture

- The instruction address bus and instruction bus used for reading instructions from memory. The address bus and data bus are used for writing and reading data to and from memory.

Comparison



Von Neumann Architecture	Harvard Architecture
1. It requires less hardware	1. It requires more hardware
2. Von-Neumann Architecture requires less space.	2. Harvard Architecture requires more space.
3. Von Neumann Architecture is based on the stored-program concept.	3. Harvard Architecture is based on relay-based computer models.
4. Controlling becomes simpler since either data or instructions are to be fetched at a time.	4. Controlling becomes complex since data and instructions are to be fetched simultaneously.
5. In Von Neumann Architecture, Common bus used for transferring instructions and data.	5. In Harvard Architecture, Separate buses are used to transfer instructions and data.
6. Speed of execution is slower since it cannot fetch the data and instructions at the same time.	6. Speed of execution is faster because the processor fetches data and instructions simultaneously .
7. Von Neumann Architecture is cheaper than Harvard architecture.	7. Harvard Architecture is more expensive than Von Neumann's architecture.
8. It is mainly used in personal computers (PC)	8. It is mainly used in micro-controllers and signal processing.

Microprocessor

It is a programmable device which has all the functions of CPU of a computer.

CPU performs following functions:

- Access information
- Store information
- Execute and process information
- Give result in desired form

Introduction to 8086 Microprocessor

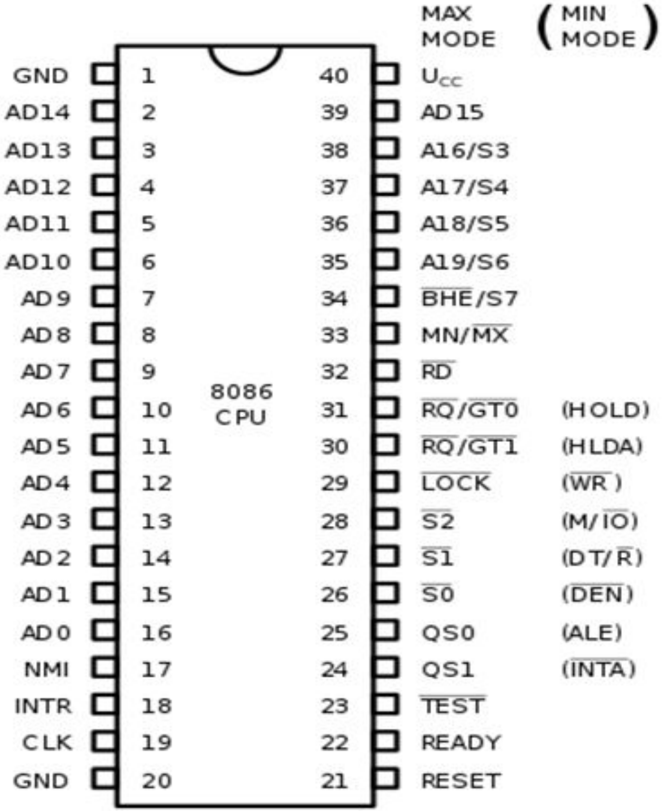
- It is a 16-bit microprocessor which means is that the ALU and the internal registers work with 16 bit of binary data at a time.
- It has 16 data lines and 20 address lines.
- It is capable to do multiplication and division also.
- It has two modes of operation: maximum mode and minimum mode.

Introduction to 8086 Microprocessor

- **Maximum mode:** In this, various control signals of microprocessor are generated by some external logic.

- **Minimum mode:** In this, various control signals of microprocessor are generated by its own.

Introduction to 8086 Microprocessor



Introduction to 8086 Microprocessor

Three types of signals available in 8086:

1. Common mode signal
2. Minimum mode signal
3. Maximum mode signal

1. Common mode signal

- **AD0-AD15 (Address Data Bus):** Bidirectional address/data lines. These are low order address bus. They are multiplexed with data. When these lines are used to transmit memory address, the symbol A is used instead of AD, for example, A0- A15.
- **A16 - A19 (Output):** High order address lines. These are multiplexed with status signals.

1. Common mode signal

- **BHE/S7 (Output):** Bus High Enable/Status. During T1, it is low. It enables the data onto the most significant half of data bus, D8-D15. 8-bit device connected to upper half of the data bus use BHE signal. It is multiplexed with status signal S7. S7 signal is available during T3 and T4.

1. Common mode signal

- **RD (Read):** For read operation. It is an output signal. It is active when LOW.
- **Ready (Input):** The addressed memory or I/O sends acknowledgment through this pin. When HIGH, it denotes that the peripheral is ready to transfer data.
- **RESET (Input):** System reset. The signal is active HIGH.
- **CLK (input):** Clock 5, 8 or 10 MHz.

1. **Common mode signal**

- **INTR:** Interrupt Request.
- **NMI (Input):** Non-maskable interrupt request.
- **TEST (Input):** Wait for test control. When LOW the microprocessor continues execution otherwise waits.
- **VCC:** Power supply +5V dc.
- **GND:** Ground.

2. Minimum mode signal

- **INTA (Output):** Pin number 24 interrupts acknowledgement. On receiving interrupt signal, the processor issues an interrupt acknowledgment signal. It is active LOW.
- **ALE (Output):** Pin no. 25. Address latch enable. It goes HIGH during T1. The microprocessor 8086 sends this signal to latch the address into the Intel 8282/8283 latch.

2. Minimum mode signal

- **DEN (Output):** Pin no. 26. Data Enable. When Intel 8287/8286 octal bus transceiver is used this signal. It is active LOW.
- **DT/R (output):** Pin No. 27 data Transmit/Receives. When Intel 8287/8286 octal bus transceiver is used this signal controls the direction of data flow through the transceiver. When it is HIGH, data is sent out. When it is LOW, data is received.

2. Minimum mode signal

- **M/IO (Output):** Pin no. 28, Memory or I/O access. When this signal is HIGH, the CPU wants to access memory. When this signal is LOW, the CPU wants to access I/O device.
- **WR (Output):** Pin no. 29, Write. When this signal is LOW, the CPU performs memory or I/O write operation.

3. Maximum mode signal

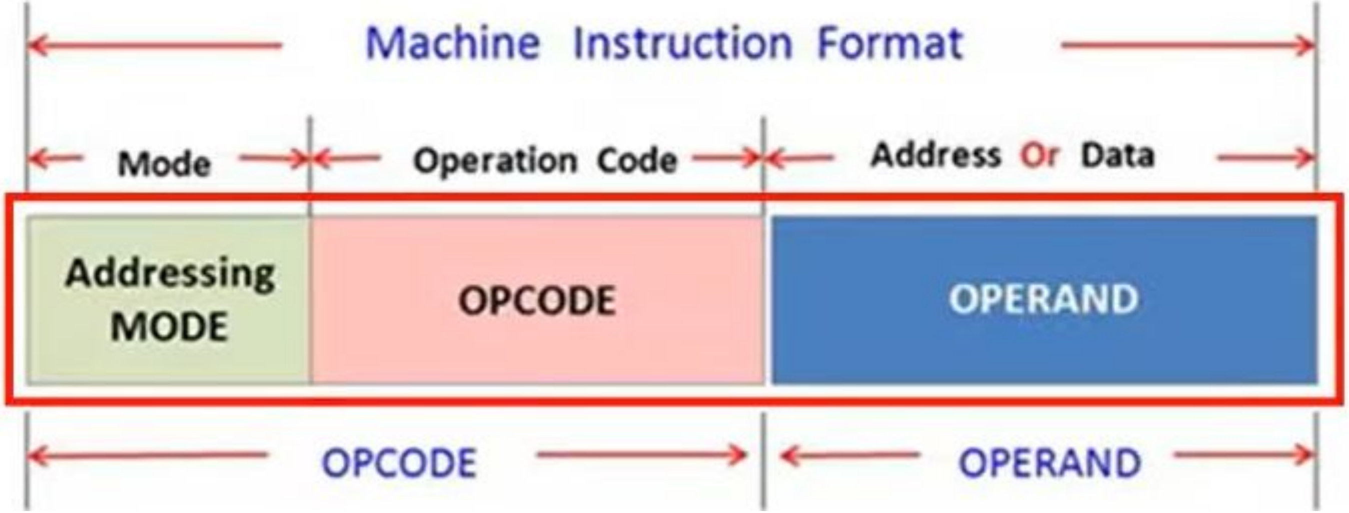
- **S0, S1, S2 (Output):** Pin numbers 26, 27, 28 Status Signals. These signals are connected to the bus controller of Intel 8288 which generates memory and I/O access control signals.
- **LOCK (Output):** Pin no. 29. It is an active LOW signal. When this signal is LOW, all interrupts are masked and no HOLD request is granted.
- **RG/GT1, RQ/GT0 (Bidirectional):** Pin numbers 30, 31, Local Bus Priority Control. Other processors ask the CPU by these lines to release the local bus.

2. Minimum mode signal

- **HLDA (Output):** Pin no. 30, Hold Acknowledgment. It is sent by the processor when it receives HOLD signal. It is active HIGH signal. When HOLD is removed HLDA goes LOW.
- **HOLD (Input):** Pin no. 31, Hold. When another device in microcomputer system wants to use the address and data bus, it sends HOLD request to CPU through this pin. It is an active HIGH signal.

Addressing modes

The term addressing modes refers to the way in which the operand of an instruction is specified. It specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.



Addressing modes

The 8086 microprocessors have 8 addressing modes.

Two addressing modes have been provided for instructions which operate on register or immediate data.

1. Register Addressing

In this, the operands / values are stored within any of the internal registers itself. So, the processing on these operands is done either in those registers or by shifting their values to some other registers. The operand is placed in one of the 16-bit or 8-bit general purpose registers.

Example

- MOV AX, CX
- ADD AL, BL
- ADD CX, DX

2. Immediate Addressing

In this addressing mode, the operands are specified within the instructions which means is that the instruction will either contain the values itself or will contain the operands whose values are required.

Example

- MOV AL, 35H
- MOV BX, 0301H
- MOV [0401], 3598H

3. Direct Addressing

In direct addressing mode, the operands offset is given in the instruction as an 8-bit or 16-bit displacement element.

Example

- ADD AL, [0301]

The instruction adds the content of the offset address 0301 to AL. the operand is placed at the given offset (0301) within the data segment DS.

4. Register Indirect Addressing

The operand's offset is placed in any one of the registers BX, BP, SI or DI as specified in the instruction.

Example

- MOV AX, [BX]

It moves the contents of memory locations addressed by the register BX to the register AX.

5. Based Addressing

The operand's offset is the sum of an 8-bit or 16-bit displacement and the contents of the base register BX or BP. BX is used as base register for data segment, and the BP is used as a base register for stack segment.

Effective address (Offset) = [BX + 8-bit or 16-bit displacement].

Example

- MOV AL, [BX+05]; an example of 8-bit displacement.
- MOV AL, [BX + 1346H]; example of 16-bit displacement.

6. Indexed Addressing

The offset of an operand is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.

Offset (Effective Address) = [SI or DI + 8-bit or 16-bit displacement]

Example

- MOV AX, [SI + 05]; 8-bit displacement.
- MOV AX, [SI + 1528H]; 16-bit displacement.

7. Base Indexed Addressing

The offset of operand is the sum of the content of a base register BX or BP and an index register SI or DI.

Effective Address = [BX or BP] + [SI or DI]

Here, BX is used for data segment, and BP is used for stack segment.

Example

- ADD AX, [BX + SI]
- MOV CX, [BX + SI]

8. Base Indexed with displacement Addressing

In this mode of addressing, the operand's offset is given by:

Effective Address (Offset) = [BX or BP] + [SI or DI] + 8-bit or 16-bit displacement

Example

- MOV AX, [BX + SI + 05]; 8-bit displacement
- MOV AX, [BX + SI + 1235H]; 16-bit displacement

Other Addressing modes:

- **Program Memory Addressing Mode:** These types of addressing modes are used in branch instructions like JMP or CALL instruction.

  Example: JMP 0006H
- **Stack Memory Addressing Mode:** The stack memory addressing mode is used whenever you perform a push or pop operation.

  Example: PUSH BX

Instruction Set of 8086

Instructions are classified on the basis of functions they perform. They are categorized into the following main types:

- Data Transfer instruction
- Arithmetic Instructions
- Logical Instructions
- Rotate Instructions
- String Instructions

Instruction Set of 8086

The 8086 microprocessor supports 8 types of instructions −

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand.

Instruction to transfer a word

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.
- **XCHG** – Used to exchange the data from two locations.
- **XLAT** – Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** – Used to read a byte or word from the provided port to the accumulator.
- **OUT** – Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- **LEA** – Used to load the address of operand into the provided register.
- **LDS** – Used to load DS register and other provided register from the memory
- **LES** – Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- **LAHF** – Used to load AH with the low byte of the flag register.
- **SAHF** – Used to store AH register to low byte of the flag register.
- **PUSHF** – Used to copy the flag register at the top of the stack.
- **POPF** – Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Instructions to perform addition

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.
- **AAM** – Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** – Used to divide the signed word by byte or signed double word by word.
- **AAD** – Used to adjust ASCII codes after division.
- **CBW** – Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** – Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Instructions to perform logical operation

- **NOT** – Used to invert each bit of a byte or word.
- **AND** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** – Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** – Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

- **ROL** – Used to rotate bits of byte/word towards the left, i.e.MSB to LSB and to Carry Flag [CF].
- **ROR** – Used to rotate bits of byte/word towards the right, i.e.LSB to MSB and to Carry Flag [CF].
- **RCR** – Used to rotate bits of byte/word towards the right, i.e.LSB to CF and CF to MSB.
- **RCL** – Used to rotate bits of byte/word towards the left, i.e.MSB to CF and CF to LSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

- **REP** – Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** – Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** – Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** – Used to compare two string bytes/words.
- **INS/INSB/INSW** – Used as an input string/byte/word from the I/O port to the provided memory location.
- **OUTS/OUTSB/OUTSW** – Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** – Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** – Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution.

Instructions to transfer the instruction during an execution without any condition −

- **CALL** − Used to call a procedure and save their return address to the stack.
- **RET** − Used to return from the procedure to the main program.
- **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions

- JA/JNBE – Used to jump if above/not below/equal instruction satisfies.
- JAE/JNB – Used to jump if above/not below instruction satisfies.
- JBE/JNA – Used to jump if below/equal/ not above instruction satisfies.
- JC – Used to jump if carry flag CF = 1
- JE/JZ – Used to jump if equal/zero flag ZF = 1
- JG/JNLE – Used to jump if greater/not less than/equal instruction satisfies.
- JGE/JNL – Used to jump if greater than/equal/not less than instruction
- JL/JNGE – Used to jump if less than/not greater than/equal instruction
- JLE/JNG – Used to jump if less than/equal/if not greater than instruction
- JNC – Used to jump if no carry flag (CF = 0)
- JNE/JNZ – Used to jump if not equal/zero flag ZF = 0
- JNO – Used to jump if no overflow flag OF = 0
- JNP/JPO – Used to jump if not parity/parity odd PF = 0
- JNS – Used to jump if not sign SF = 0
- JO – Used to jump if overflow flag OF = 1
- JP/JPE – Used to jump if parity/parity even PF = 1
- JS – Used to jump if sign flag SF = 1

Processor Control Instructions

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

- **STC** – Used to set carry flag CF to 1
- **CLC** – Used to clear/reset carry flag CF to 0
- **CMC** – Used to put complement at the state of carry flag CF.
- **STD** – Used to set the direction flag DF to 1
- **CLD** – Used to clear/reset the direction flag DF to 0
- **STI** – Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** – Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times.

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** − Used to jump to the provided address if CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1
- **IRET** – Used to return from interrupt service to the main program