**Chapter 2: x86 PROCESSOR ARCHITECTURE**

**Topic – 1: General Concepts**

**Introduction**

* x86 microprocessors are used in both **Intel** & **AMD** processors.
* This includes **Intel IA-32** & **Intel 64** processors like **Intel Pentium** & **Core-Duo**.
* **AMD:** Advanced micro devices.
* It also includes AMD processors such as **Athlon**, **Phenom**, **Opteron** & **AMD64**.

**Basic Microcomputer Design**

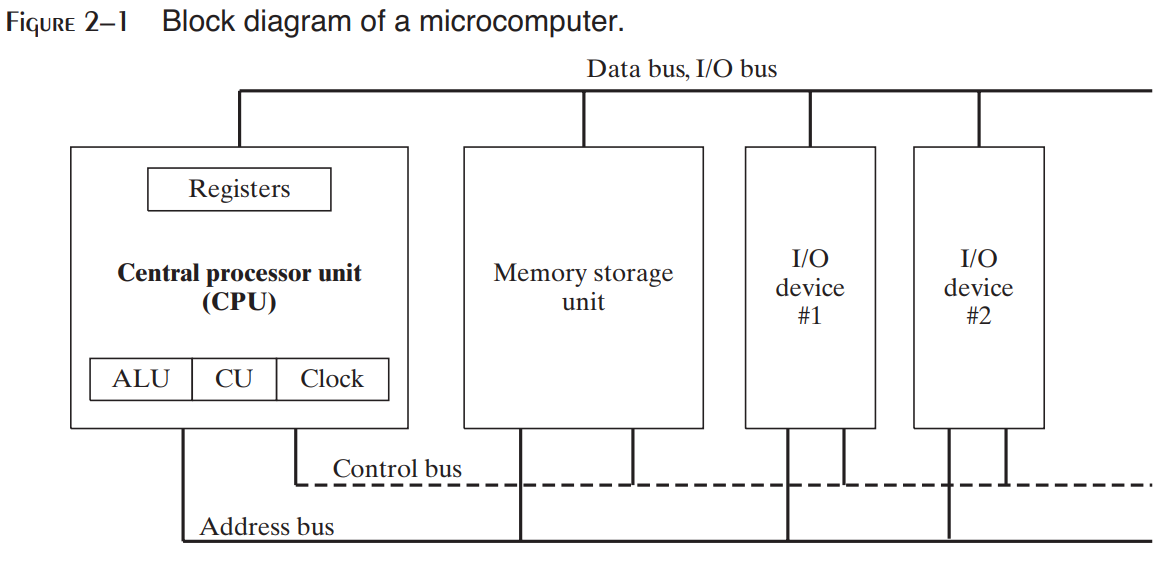
* CPU contains: **Registers, high-frequency clock, a CU (control unit) & ALU.**
* **CU:** Puts machine instructions to be executed in a meaningful sequence.
* **ALU:** Performs **arithmetic** & **logical** (AND, OR, NOT etc) operations.
* CPU is attached to whole computer via its **pins** which are connected to **socket** present in the motherboard.
* Its most pins are connected to the **data bus**, **control bus** & **address bus**.

**Data Transfer Mechanism**

* **Memory storage unit:** Stores the **instructions** & **data** of a running program.
* HDD & SSD etc are type of **memory storage unit**.
* For execution of a program, it is first loaded to **RAM**.
* CPU requests **storage unit** for data.
* Programs are then copied from **RAM** to **CPU** for being processed.
* And finally, data are transferred from **CPU** to **storage unit**.

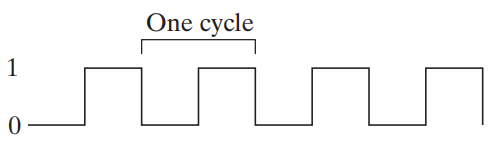
**Buses**

* **Bus:** Group of **parallel wires** that transfer data from one part to another in computer.
* Types of buses: **Data, I/O, control & address.**
* **Data bus:** Transfers instructions & data between **CPU** & **memory**.
* **I/O bus:** Transfers data between **CPU** and **peripheral** **devices**.
* **Control bus:** **Synchronizes** all devices connected to bus system by **using binary signals**.
* **Address bus:** Contains addresses of instructions & data being **transferred** between CPU & memory.



**Clock**

* **Synchronizes** CPU & system bus when they are active.
* Pulses at a constant rate.
* **Machine cycle/ clock cycle:** Unit of time for measuring machine instructions.



* **One clock cycle** is the time required for **one clock pulse** to complete.

**Duration of clock cycle = Reciprocal of clock’s speed**

**Example**

**A clock that oscillates 1 billion times per second (clock speed of 1 GHz), has a clock cycle of 1 billionth of a second (1 nano second).**

* One machine instruction requires **atleast** oneclock cycle to execute.
* Some instructions require **more than 50** clock cycles to execute.
* Like multiply instruction in 8088 processor.
* **Wait states:** Empty clock cycles.
* It is called **wait state** due to **unsynchronized** CPU, system bus & memory circuits for that duration.
* Instructions requiring **memory access** often go through **wait states**.
* Clock pulses are required mainly for **executing** instructions.
* That’s why when accessing memory, clock pulses are temporarily halted (clock cycle).

**Instruction Execution Cycle**

* It is a **predefined procedure** CPU goes through for executing a machine instruction.
* **Instruction pointer register:** Contains the address of the instruction to be executed.
* The **bit pattern** of an instruction can also tell us what operands (inputs) it has.

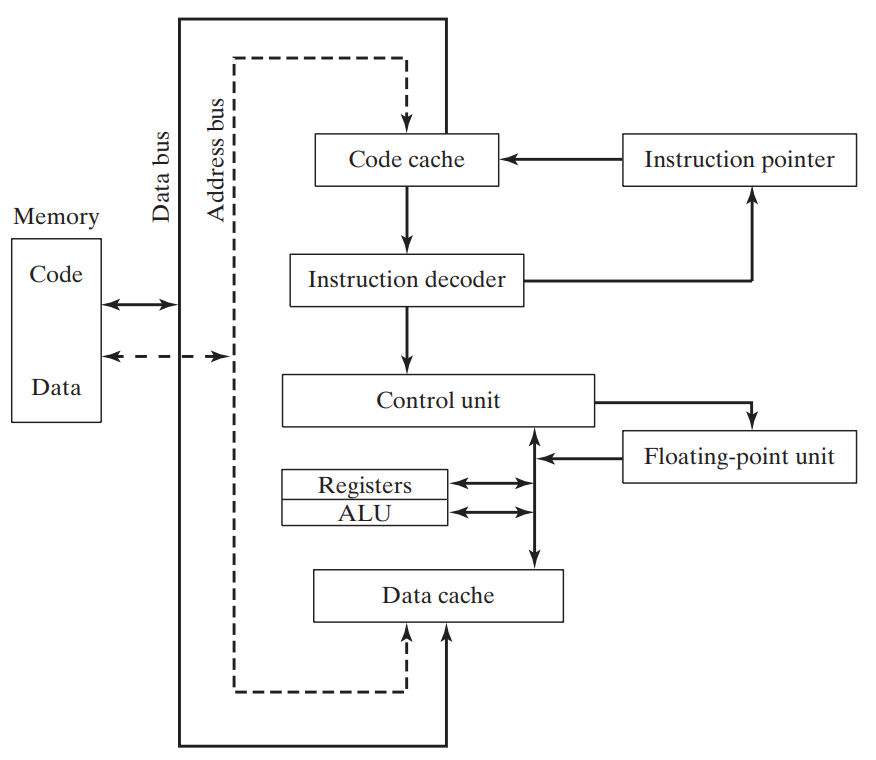
**Topic – 2: Steps for Instruction Execution**

**Steps In Brief**

* **Step 1:** CPU fetches instruction from **instruction queue**.
* **Step 2:** CPU increments the **instruction pointer**.
* **Step 3:** CPU decodes the instruction by reading its binary **bits signature**.
* **Step 4:** If **operands** are involved, they are **fetched** from registers & memory.
* **Step 5:** CPU executes the instruction using those operands.
* **Step 6:** CPU updates some **status flags** (zero, overflow, carry etc).
* **Step 7:** If an **output operand** is involved, then CPU stores output result in it.

**In Nutshell**

* In short – **fetch**, **decode** & **execute**.
* **Z = X + Y** [**X** & **Y** are **input** operands, **Z** is **output** operand]



**Reading Instructions From Memory**

* Computer reads memory slower than accessing registers.
* This is because **4 steps** are involved when reading a value from memory.

**Steps In Brief - II**

* **Step 1:** Address is sent through address bus.
* **Step 2:** Change the processor’s **RD** (**read**) pin’s status.
* **Step 3:** Then memory chips take **one** **clock cycle** to respond.
* **Step 4:** Copy of requested data is sent through **data bus** to the **destination operand**.

**About Steps**

* Each of these steps take **one clock cycle** each.
* Whereas the **CPU registers** are accessed in **one clock cycle**.
* To solve this issue, the CPU designers created **memory *cache***.
* **Cache** is a **fast** access memory which stores **recently used** instructions.
* This is because recently used instructions are **expected** to be used again soon.
* **Cache** stores both **memory** of the instruction & the **codes** of instructions.
* **Cache hit:** CPU **finds** something it was finding in **cache**.
* **Cache miss:** CPU **doesn’t** **find** something it was finding in **cache**.

**Types Of Cache**

* **Level – I cache:** Also known as **primary cache** & stored on **CPU** itself.
* **Level – II cache:** Also known as **secondary cache** & stored **near the CPU**.
* **Level – II cache** is little **slower** than **level – I cache** & attached to CPU through a high-speed **bus**.
* **Cache** is **faster** than **conventional RAM** because it uses ***static RAM***.
* **Conventional RAM:** Dynamic RAM
* **Static RAM** is faster than **dynamic RAM** because it holds its contents **without** being refreshed.
* Thus, its **expensive** too.

**Loading & Executing Program**

* ***Program loader*** loads the program into memory.
* Then the OS points CPU to the **entry point** of our program.
* The **address** of **entry point** is from where the execution has to start.
* But there are **more** steps involved in it.

**Steps In Brief - III**

* **Step 1:** OS **searches** for the program using its **filename** in the current & surrounding directories.
* **Step 2:** Then the OS **finds location** of **next free block** of memory in RAM.
* **Step 3:** When found, information like its **file size** & **physical location** on disk are fetched.
* This block is called ***descriptor table***.
* **Step 4:** Then a ***process ID*** is given to the running program (**process**).

***\*It is OS’s responsibility to manage resources requested by the instruction & keep track of the program\****

**Task Manager Processes (Windows)**

* Applications processes
* Background processes
* Windows processes

**Topic – 3: 32-Bit x86 Processors**

**Included Processors**

* It includes **32-bit** processors of both **Intel** and **AMD**.

**Modes Of Operation**

* Protected mode
* Real-address mode
* System management mode (SMM)
* Virtual-8086 mode

**Modes Description**

* First three modes are **primary modes** & last one is **secondary mode**.

**Protected Mode**

* It is the **default state** of the processor.
* Programmer is free to use **all features** here.
* Each program runs in **separate memory spaces** called ***segments***.
* Processor avoids each program from **crossing** their **segment**.
* This ensures that any trouble in a program **doesn’t** affect other processes running at the **same time** by entering its memory space.

**Virtual-8086 mode**

* Same as **protected mode** but it has ability to run **multiple sessions** parallelly.

**Real-Address Mode**

* Provides an **early** **Intel** processor environment to write instructions.
* But with some **extra features**, like **switching** to **other modes** when within this mode.
* Now an advantage of using **old Intel** interface is that it provides **direct access** to memory & devices.

**System Management Mode (SMM)**

* Allows programmers to implement features like **power management** & **system security**.
* And these are used by **computer manufacturers**.

**Topic – 4: Basic Execution Environment**

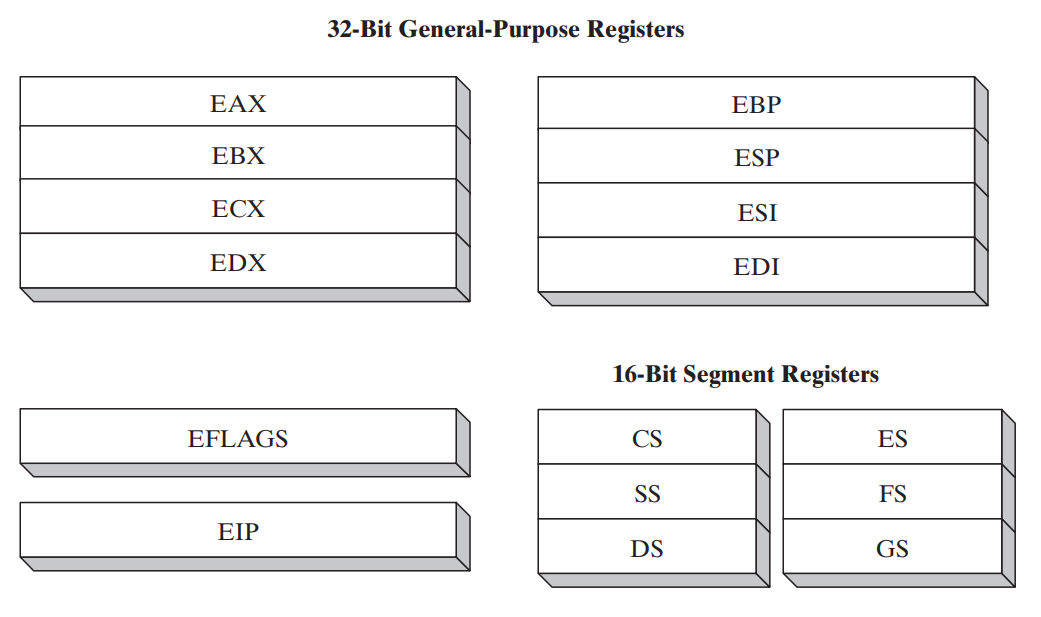
**Address Space**

* In **protected mode**, a program can address up to **4 GB** of linear space.
* However, a technique called ***extended physical addressing*** in **P6 processor** allows up to **64 GB** of linear addressing.
* Whereas in **real-addressing mode**, a program can address only up to **1 MB** of linear space.
* So, in **virtual-8086 mode** each session can address up to **1 MB** of linear space.

**Topic – 5: Basic Program Execution Registers**

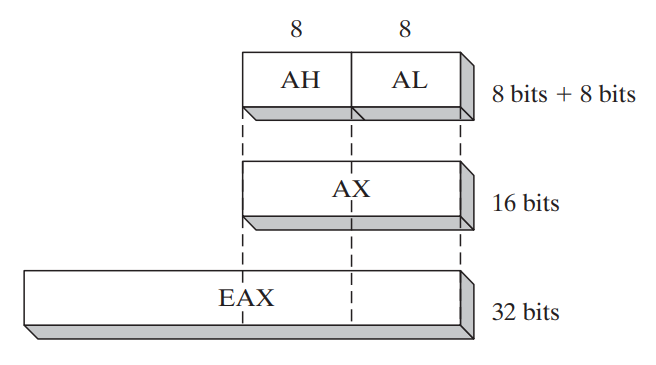
**Introduction**

* ***Registers*** are storage locations which are **faster** than **memory**.
* So, when **loops** are optimized, they are stored in **registers** instead **variables**.
* There are **8** ***general-purpose registers***, **6** ***segment registers***, a ***status flag register*** & an ***instruction pointer***.



**General Purpose Registers (GPR)**

* Primarily used for **arithmetic** & **data movements**.
* The **lower** **16-bits** of **EAX** register is referenced by name **AX**.



* We can access the lower & higher **8-bits** of lower **16-bits** in half of the **general-purpose registers**.
* The higher **8-bits** of **AX** is referenced as **AH** & lower **8-bits** are as **AL**.

|  |  |  |  |
| --- | --- | --- | --- |
| **32-Bit** | **16-Bit** | **8-Bit (High)** | **8-Bit (Low)** |
| **EAX** | **AX** | **AH** | **AL** |
| **EBX** | **BX** | **BH** | **BL** |

* We can access lower **16-bits** of rest half of the **general-purpose registers**.

|  |  |
| --- | --- |
| **32-Bit** | **16-Bit** |
| **ESI** | **SI** |
| **EDI** | **DI** |
| **EBP** | **BP** |
| **ESP** | **SP** |

**Topic – 6: Specialized Uses Of GPRs**

**EAX**

* Automatically used for **multiplication** & **division**.
* Also called ***extended accumulator*** register.

**ECX**

* Automatically used as a **loop counter**.

**ESP**

* Addresses data in **stack**.
* **Stack:** A type of memory structure in system.
* It is **not** used much in **arithmetic** & **data transfer** operations.
* Also called ***extended stack pointer*** register.

**ESI**

* Used in **high-speed** memory transfers.
* Also called ***extended source index*** register.

**EDI**

* It is also used in **high-speed** memory transfers.
* Also called ***extended destination index*** register.

**EBP**

* Used by **high-level** languages to reference **function parameters** & **local variables** on **stack**.
* It is advised **not** to be used in **ordinary** arithmetic & data transfers.
* Also called ***extended frame pointer*** register.

**Topic – 7: More Types Of Registers**

**Segment Register**

* ***Segment registers*** are of **16-bits**.
* In **real-addressing** mode, they hold values of real **base addresses** of some memory segments.
* In **protected mode**, they hold **pointers** to ***segment descriptor tables***.
* Some of it hold the **instructions** (code), some hold **variables** (data) & ***segment stack*** register holds local **function’s** **variables** and **parameters**.

**Instruction Pointer**

* **EIP** is ***instruction pointer*** register.
* It contains the address of **next instruction** to be executed.
* We can manipulate **EIP** through code in order to branch program at **new location**.

**EFLAGS Register**

* Contains individual **bits** that can manipulate **operations of CPU** & also tell the **nature** of some operations.
* We call flag as **set** when it is **1** & **clear** when it is **0**.

**Topic – 8: EFLAGS Register (Brief)**

**Control Flags**

* Can controls the CPU’s operations.
* Example 1: **Breaking** & **interrupting** when **out of range value** is detected.
* Example 2: Entering different modes like **virtual-8086** or **protected mode** etc.
* Flags like **direction** and **interrupt** can be set in **EFLAGS** register.

**Status Flags**

* **Carry flag (CF):** **Set** when result of an **unsigned arithmetic operation** is **too large** to fit at the destination.
* **Overflow flag (OF):** **Set** when result of a **signed arithmetic operation** is **too large** or **too small** to fit at the destination.
* **Sign flag (SF):** **Set** when result of an operation is **negative**.
* **Zero flag (ZF):** **Set** when result of an operation is **zero**.
* **Auxiliary carry (AC):** In an **8-bit** operator, it is set when an arithmetic operation generates a **carry** on the **4th bit** from **3rd bit**.
* **Parity flag (PF):** Set when ***least significant******byte*** in a result contains **even** number of **1s**.

**Topic – 9: MMX Registers**

* ***MMX*** registers are of **64-bit** each.
* **MMX** is a technology that has enhanced performance of **Intel** processors.
* They are used in **advanced multimedia** & **communication** applications.
* There are **eight MMX** registers which support special instructions called ***SIMD***.
* **SIMD:** Single-instruction, multiple-data
* **MMX** registers can operate on **multiple values** in it **parallelly**.
* Same registers are used by ***floating-point unit*** (FPU).

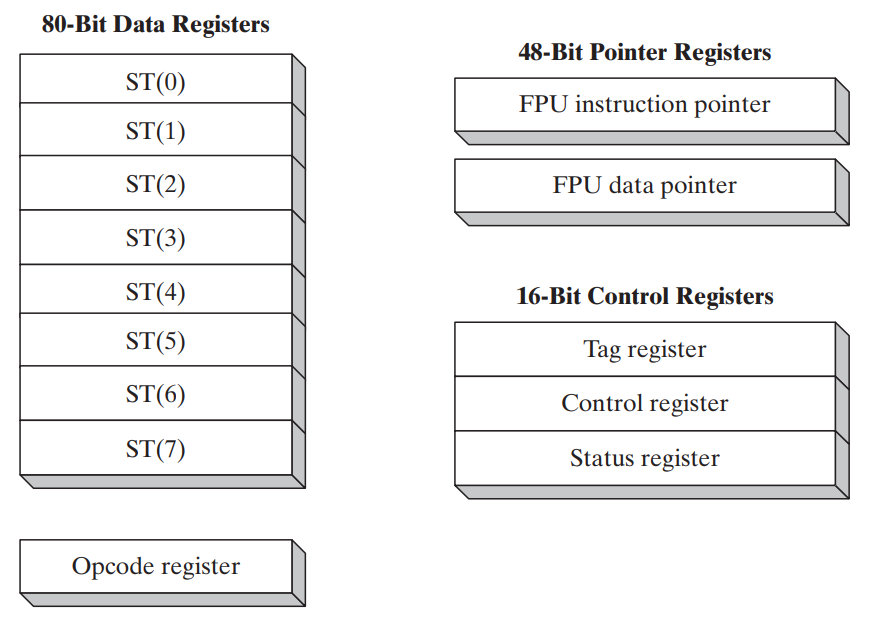
**Topic – 10: XMM Registers**

**Introduction**

* These are set of **eight** **128-bit** registers.
* They can be used by connecting **SIMD** extensions to the **instruction set**.

**Floating-Point Unit (FPU)**

* Does **high-speed** floating-point arithmetic.
* Earlier separate chip was required for it, but now it is attached to the main processor.
* There are **eight** registers in **FPU**, from **ST(0)** to **ST(7)**.
* Rest of them are **control** & **pointer** registers.



**Topic – 11: x86 Memory Management**

**Introduction**

* Processor’s way of managing memory **varies** from mode to mode.
* **Protected mode** is the **most powerful** mode & it **restricts** application programs from directly accessing hardware.
* In **real-addressing mode**, **20 MB** of memory can be addressed **(00000)16** to **(FFFFF)16**.
* And **application programs** are free to access **any** location, even those linked directly to **system hardware**.
* **MS-DOS** runs in this mode, while **Windows 95** and **98** can switch to it by **booting**.

**Nuisance Around Protected Mode**

* In **protected-mode**, programs **can’t** access each other’s **data**.
* **Modern Windows** and **Linux** run in **protected mode**.
* In **virtual-8086 mode**, **Windows NT** and **2000** had ***Command window*** which allowed users to run **virtual-8086 machine**.
* While in those OSes, some programs using **MS-DOS** **won’t** run.

**Topic – 12: 64-Bit x86-64 Processors**

**x86 Instruction Set Features**

* Addresses are **64-bit** long with **264 bytes** of ***virtual address space***.
* **General-purpose registers** in x86 are of **32-bit** as we saw.
* So, **instruction set** uses **32-bit extension** of those registers making them of **64-bit**.
* Thus, they can have **64-bit** **integer** **operands**.
* Also, **x86 instruction set** can use **48-bit** of ***physical address space***.
* We **can’t** run **16-bit real mode** or **virtual-8086 mode** on native **64-bit mode**.
* The **16-bit real mode** can be run through a ***legacy mode*** though but it is **not** available in **64-bit Windows**.

***\*x86-64 is an instruction set, but we will treat it as a processor type onwards\****

**Topic – 13: 64-Bit Operation Mode**

**Introduction**

* **Intel** **64** architecture has a new mode called ***IA-32e***.
* **IA-32e** again has two **sub-modes**:
  + ***Compatibility mode***
  + ***64-bit mode***
* We will refer these **sub-modes** as **modes** from now onwards.

**Compatibility Mode**

* Existing **16-bit** & **32-bit** applications can run **without** recompilation.
* However, **Win16** and **DOS** applications **won’t** run in **64-bit Windows**.
* **Win16:** 16-bit Windows applications.
* Moreover, **64-bit Windows** also **don’t** have ***virtual DOS machine subsystem***, which can run Windows **DOS** applications in **virtual-8086 mode**.

**64-Bit Mode**

* Runs applications using **64-bit linear address space**.
* Native mode for **64-bit Windows**.
* This mode even enables programmers to use **64-bit operands** in instructions.

**Topic – 14: Basic 64-bit Execution Environment**

**Introduction**

* In **64-bit mode**, processors **theoretically** support **64-bit long addresses**.
* But **practically**, they can use **48-bit long addresses** right now.

**64-Bit Registers v/s 32-Bit Registers**

|  |  |
| --- | --- |
| **64-bit Processor** | **32-bit Processor** |
| **Sixteen 64-bit general-purpose registers.** | **Eight 32-bit general-purpose registers.** |
| **Eight 80-bit floating point registers.** | *-SAME-* |
| **64-bit status flag is called RFLAGS.** | **32-bit status flag is called EFLAGS.** |
| **Only lower 32-bit of status flag are used.** | **Only lower 17-bit of status flag are used.** |
| **64-bit instruction pointer is called RIP.** | **32-bit instruction pointer is called EIP.** |
| **Eight 64-bit MMX registers.** | *-SAME-* |
| **Sixteen 128-bit XMM registers.** | **Eight 128-bit XMM registers.** |

**General-Purpose Registers**

* **General-purpose registers** can access **16-bit**, **32-bit** & **64-bit** operands with a special prefix.
* By default, the size of operand in **64-bit mode** is of **32-bit**.
* But when we add **REX** prefix, we can have **64-bit long** operands & total of **sixteen** **general-purpose registers** available to us.
* **Eight** of them are **same** as they were, while the **extra eight** are named **R8** to **R15**.

**Available Registers With Enabled REX**

* We are referring to only 64-bit modes here.

|  |  |
| --- | --- |
| **Operand Size** | **Available Registers** |
| **8-bit** | **AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L** |
| **16-bit** | **AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W** |
| **32-bit** | **EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D** |
| **64-bit** | **RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15** |

* In **64-bit mode**, a single **can’t** access a **high-byte** register (like **AH**) & a **low-byte** from any extra registers (like **DIL**).
* And in same **64-bit mode**, **EFLAGS** is replaced by **RFLAGS** sharing **same** **lower 32-bit**.
* Rest of the **32-bits** **aren’t** used & thus **reserved**.

**Topic – 13: Components Of x86 Computers**

**Motherboard Components**

* CPU
* Supporting processors (chipset)
* Main memory
* I/O connectors
* Power supply connectors
* Expansion slots

***\*These components are connected to each other using bus (wires)\****

**Expected Components**

* CPU socket
* Memory slots (plug-in memory boards)
* BIOS (contains system software)
* CMOS RAM (with circular battery)
* Large storage device connectors (like HDD, CD-ROMs etc)
* USB connectors
* Keyboard & mouse ports
* PCI board connectors (for sound card & graphic card etc)

**Optional Components**

* Sound processor
* Parallel & serial device connectors
* Network adapter
* AGP bus connector (for high-speed video card)

**Important Supporting Processors**

* **Floating-point unit (FPU)**
* **8284/82C284 clock generator –** Syncs CPU & whole computer.
* **8259A Programmable interrupt controller (PIC) –** Handles external interrupts like keyboard, system clock & hard drive.
* **8253 Programmable interface timer/counter –** Interrupts CPU 18.2 times each second to update date & time of system. Also responsible for refreshing memory because RAM can’t keep its data for more than few milliseconds.
* **8255 Programmable parallel port –** Transfers data from memory using IEEE parallel ports, they are used in devices like printers.

**PCI Express Bus Architecture**

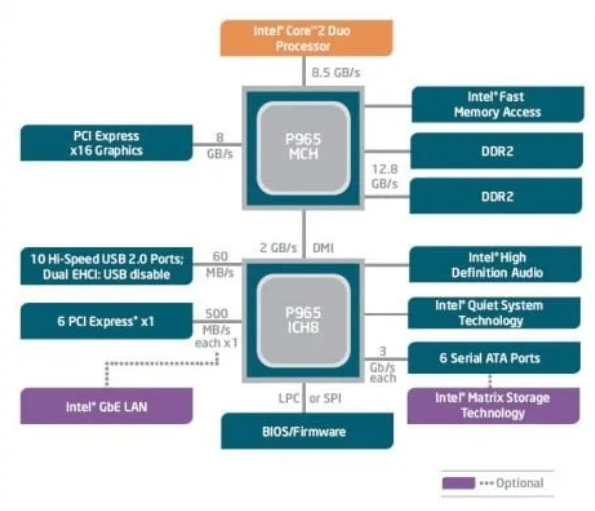
* **PCI** stands for ***peripheral component interconnect***.
* **PCI** establishes a connection between **CPU** & **other** hardware devices.
* These devices can be **hard disks**, **memory**, **network controllers** etc.
* Recent **PCIs** provide **bidirectional connections** between CPU, memory & devices.
* And they also **transfer data** in packets in **separate lanes**, at **high speeds**.

**Motherboard Chipset**

* **Motherboard chipset:** Set of chips embedded on particular type of motherboard.
* Intel **P965** is used with **Intel Core 2 Duo** or **Pentium D**.

**Features of Intel P965**

* Fast memory access uses ***memory controller hub*** (**MCH**).
* **MCH** can access **dual-channel** **DDR2** memory at **800 MHz** clock speed.
* An I/O controller hub like **Intel ICH8/R/DH** uses Intel’s **matrix storage technology** (**MST**) to support multiple hard disks.
* Supports multiple **USB ports**, **PCI express slots**, **networking** etc.
* Also has an **HD** sound chip.



**Topic – 14: Memory**

**Introduction**

* Types of memory used in Intel system are **ROM**, **EPROM**, **DRAM**, **SRAM**, **VRAM** & **CMOS** **RAM**.

**EPROM**

* Can be erased with **ultraviolet light**.

**DRAM**

* Also known as **main memory**.
* Stores data of **running programs**.
* Refreshed **every** **millisecond**.
* Some systems may use **ECC** (**error checking & correcting**) **memory** instead.

**SRAM**

* High speed cache memory.
* Expensive.

**VRAM**

* Dual-ported.
* One port **writes data** to display.
* Another port **refreshes the display** using that data.

**CMOS RAM**

* **CMOS RAM** stands for **Complimentary metal oxide semiconductor RAM**.
* Contains **system setup information** & hence **doesn’t** lose its content even when powered off.
* Refreshed using **system battery**.

**Topic – 15: Input-Output System**

**Introduction**

* We **not** necessarily need to access the hardware directly.
* This is because **OSes** provide many call functions to do so.

**Levels Of I/O Access**

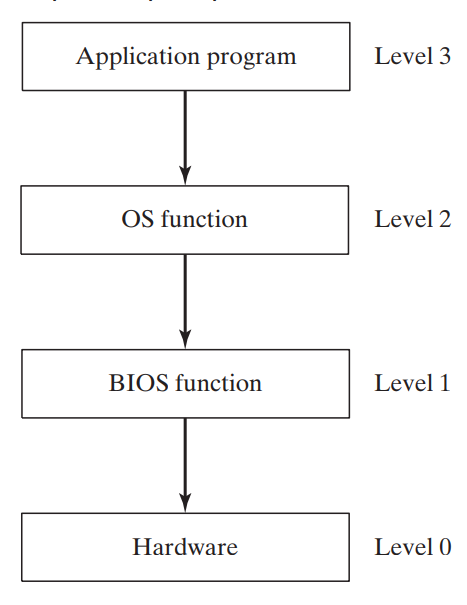
* **High-level language functions:** These are functions provided by **high-level languages** to perform I/O operations.
* **Operating system:** Provides **API libraries** to perform I/O operations like writing & reading strings from files, allocating memory blocks etc.
* **BIOS:** Contains **subroutines** that directly communicate with hardware & is installed by computer manufacturer. **OSes** usually communicate with **BIOS**.

**Device Drivers**

* They allow OS to directly communicate with hardware devices.
* These devices also include **BIOS**.
* Device drivers are either **separately installed** or installed when **device is attached**.
* In the second case, first **device signature** is detected & then its **name**.

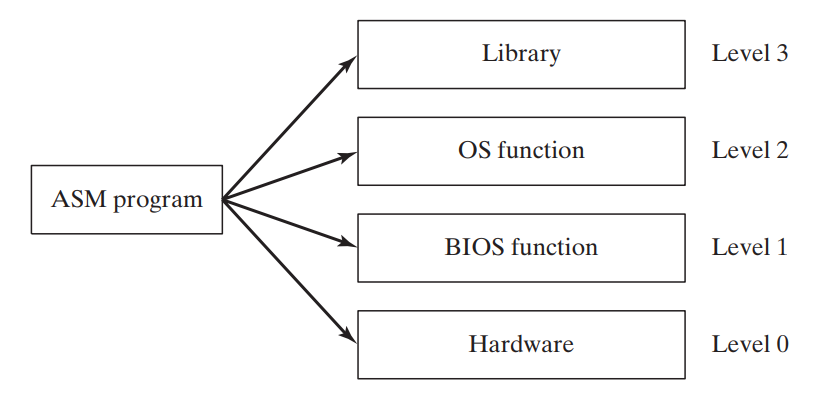
**Displaying String On Screen**

* **Step 1:** **HLL** (**high-level language**) statement **displays string** on the screen.
* **Step 2:** This function passes the **string pointer** further to an **OS function**.
* **Step 3:** This OS function further passes the **ASCII code** & **colour** of each character to a **BIOS subroutine**, using a **loop**.
* **Step 4:** Another **BIOS subroutine** is called to **advance the cursor** on screen as the characters are displayed one-by-one.
* **Step 5:** This character is then **mapped** to a **system font**.
* **Step 6:** The character is sent to a **port** attached to ***video controller card***.
* **Step 7:** **Video controller card** produces **hardware signals** to the screen which does ***raster scanning*** (lighting pixel LEDs horizontally line-by-line).



**Programming At Multiple Levels**

* **Level 3:** Using **libraries** to perform **text I/O** & **file I/O** operations.
* **Level 2:** Using **OS** to perform **text I/O** & **file I/O operations**. If an OS uses **GUI**, then it can display graphics **independent** of device.
* **Level 1:** Using **BIOS** subroutines and **manipulate** colours, graphics, sound, keyboard, disk & other hardware devices.
* **Level 0:** Receive data & specifications **directly** from hardware ports, but that makes it **unportable** because hardware like ports **may vary** across devices.



**Level 1 (BIOS)**

* Code at this level has to be different for **each system**.
* Or a programmer has to write code to **adjust** certain aspects which are **varying** across each system.
* For example, **hardware display** might be different across systems, so a programmer has to write code to adjust the **resolution** when running another system with same code.

**Level 0 (Hardware)**

* Generally, works with **generic devices**.
* **Generic devices:** Devices having many **commonness** in hardware design.
* Code at this level **directly communicates** with hardware **without** anything coming between **hardware** & **code**.
* These code needs to be modified when being **ported** to a hardware having some differences in design.
* Early video games were written at this level & thus had **full control** over the hardware.

**More Insights**

* At **BIOS level**, we can interact with hardware components directly.
* For example, we can communicate with **sound card** using its **device driver software** installed in **BIOS**.
* We can find out what **class** it belongs to & can even **fine tune** its performance as per its features for our needs.
* Though **not** many cards allow to make changes to its program.
* Most general-purpose OSes **don’t** allow users to directly access hardware in order to **avoid** making serious accidents with system.
* There are some OSes that are made for **specific devices** which directly interact with hardware, in order to keep **minimal memory occupation** rather than using complete OS.
* **MS-DOS** was **last Microsoft OS** to allow direct access to hardware.
* It was a **single-task** & **single-user** OS.

**Topic – 16: Summary**

**Terminologies**

* **BIOS:** A **collection of functions** that can directly communicate with the hardware.
* **Direction flag:** Tells the direction of processing (will know more later).
* **PCI express:** High-speed interface used to connect various components to motherboard.
* **PCI:** Peripheral Component Interface
* **PIC:** Programmable Interrupt Control
* **Programmable parallel port:** Port to allow transfer of data between computer & peripheral devices.
* **SIMD:** **Single-instruction multiple-device**, allows to execute programs involving data which are physically on **different** locations.

**Clarity To Misconceptions**

* If an arithmetic operation is **too small** or **too large** as per a given number bits, then the **overflow flag** is set. There is **nothing** like **underflow flag**.
* **REX** is used to **extend** the number of available registers for our **operand**.
* On a **32-bit** system, **FPU** is of **80-bits**.
* In **64-bit** chips, **not** all **64-bits** can be used for **addressing** because some bits are **reserved**.
* **Itanium** instruction set is completely different from **x86**.
* In **64-bit** **native** mode, you can use **16-bit real** mode but not **virtual-8086** mode.
* **x86-64** processors have more GPR than **x86** processors, like **R8**, **R9**, **R10** etc.
* Directly providing sound outputs to sound card ports can also **enhance audio quality**.