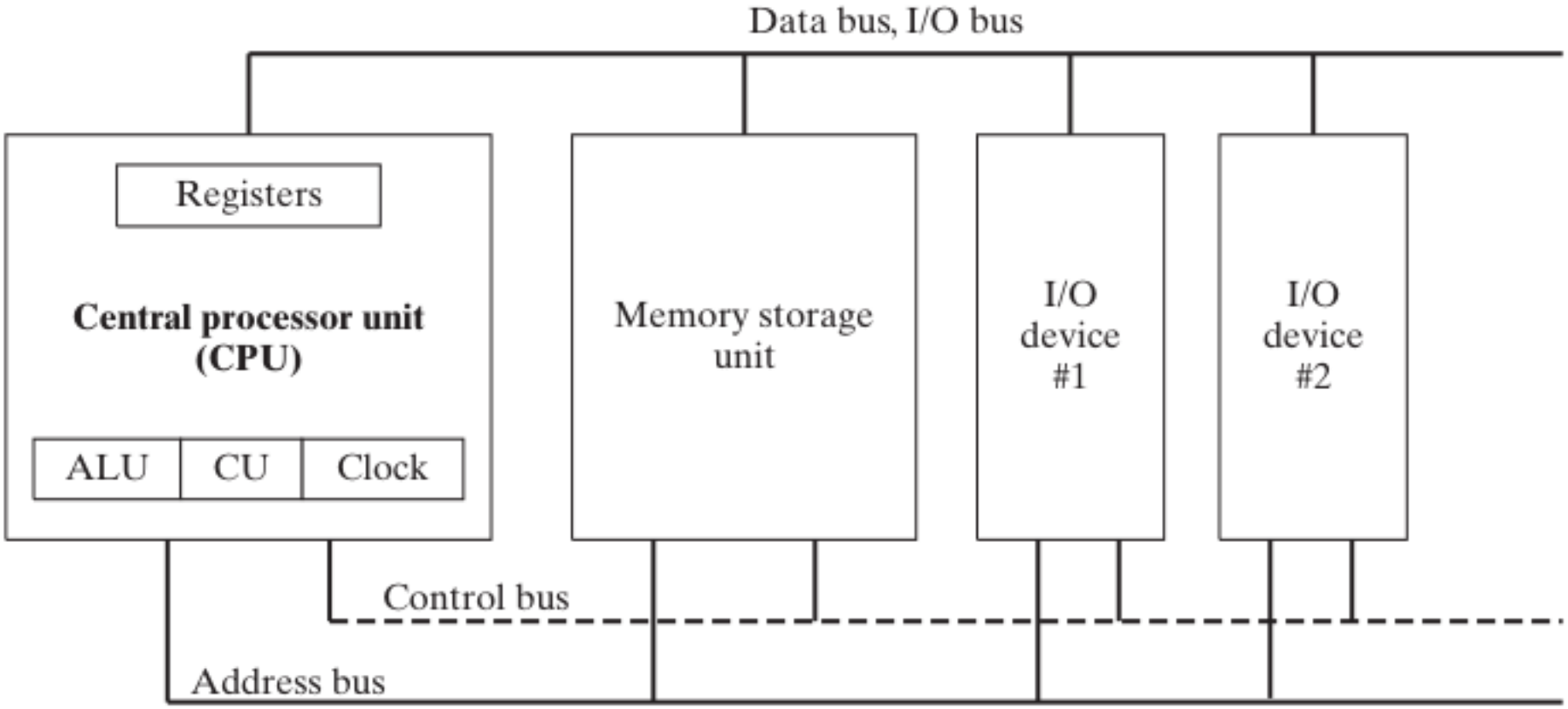
x86 PROCESSOR

Figure below shows the basic design of a hypothetical microcomputer.



The **central processor unit (CPU)**, where calculations and logical operations take place, contains a limited number of storage locations named registers, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The **clock** synchronizes the internal operations of the CPU with other system components.
- The control unit (CU) coordinates the sequencing of steps involved in executing machine

instructions.
• The arithmetic logic unit (ALU) performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer's motherboard. Most pins connect to the data bus, the control bus, and the address bus.

The **memory storage unit** is where instructions and data are held while a computer program is running.

The **storage unit** receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.

All **processing of data** takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute.

Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together.

A **bus** is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address.

The **data bus** transfers instructions and data between the CPU and memory.

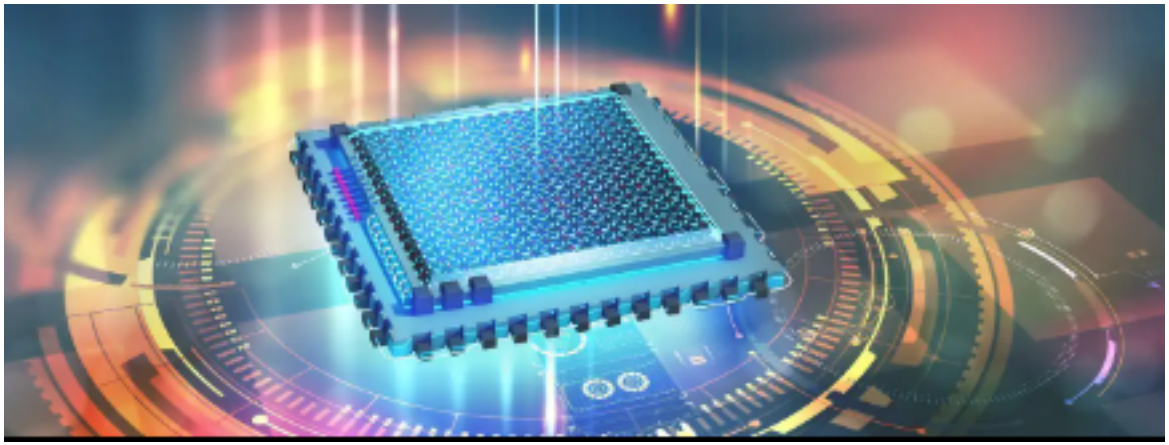
The **I/O bus** transfers data between the CPU and the system input/output devices.

The **control bus** uses binary signals to synchronize actions of all devices attached to the system bus.

The **address bus** holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

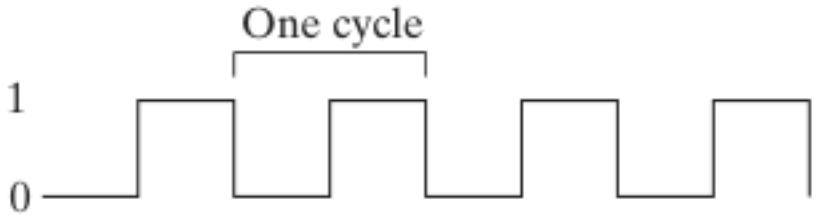
CLOCK AND CLOCK CYCLE

Sure, I can expand on this and re-write it in bullet points for easier reading:



• The CPU and system bus operations are synchronized by an internal clock pulsing at a constant rate.

• The length of a clock cycle is the time required for one complete clock pulse, measured in oscillations per second.



• The duration of a clock cycle is calculated as the reciprocal of the clock's speed. For example, a clock that oscillates 1 billion times per second (1 GHz) produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

• A machine instruction requires at least one clock cycle to execute, and some instructions require over 50 clocks (such as the multiply instruction on the 8088 processor).

• Instructions that require memory access often have empty clock cycles called wait states due to differences in the speeds of the CPU, system bus, and memory circuits.

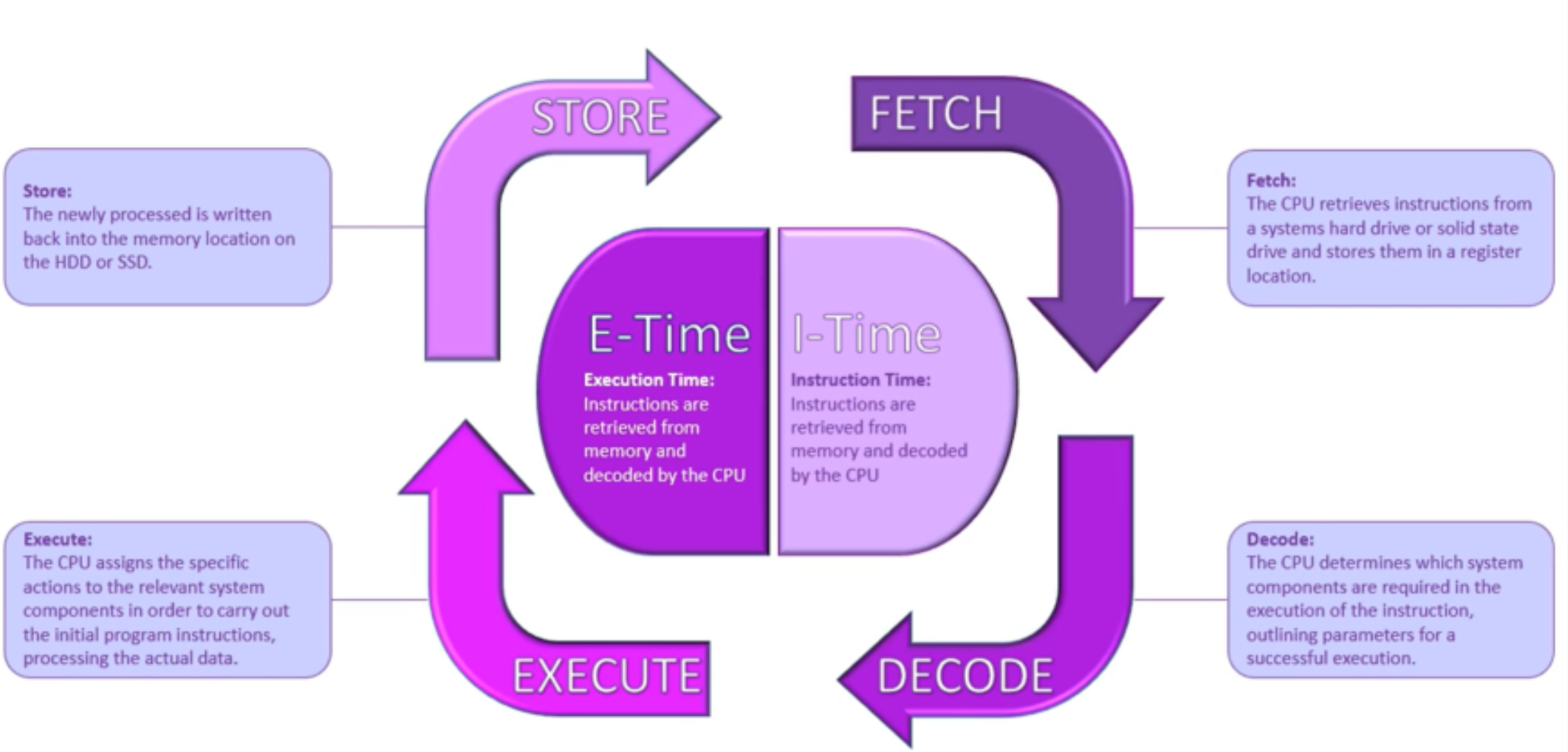
• The CPU goes through a predefined sequence of steps to execute a machine instruction, called the instruction execution cycle.

• Assuming the instruction pointer register holds the address of the instruction to execute, the steps are:

1. Fetch: The CPU fetches the instruction from memory by sending the address from the instruction pointer register to the memory circuits over the system bus.
2. Decode: The CPU decodes the instruction to determine what operation to perform and what operands to use.
3. Execute: The CPU performs the operation on the operands according to the instruction.

4. Store: The CPU stores the result of the operation back to memory or to a register.

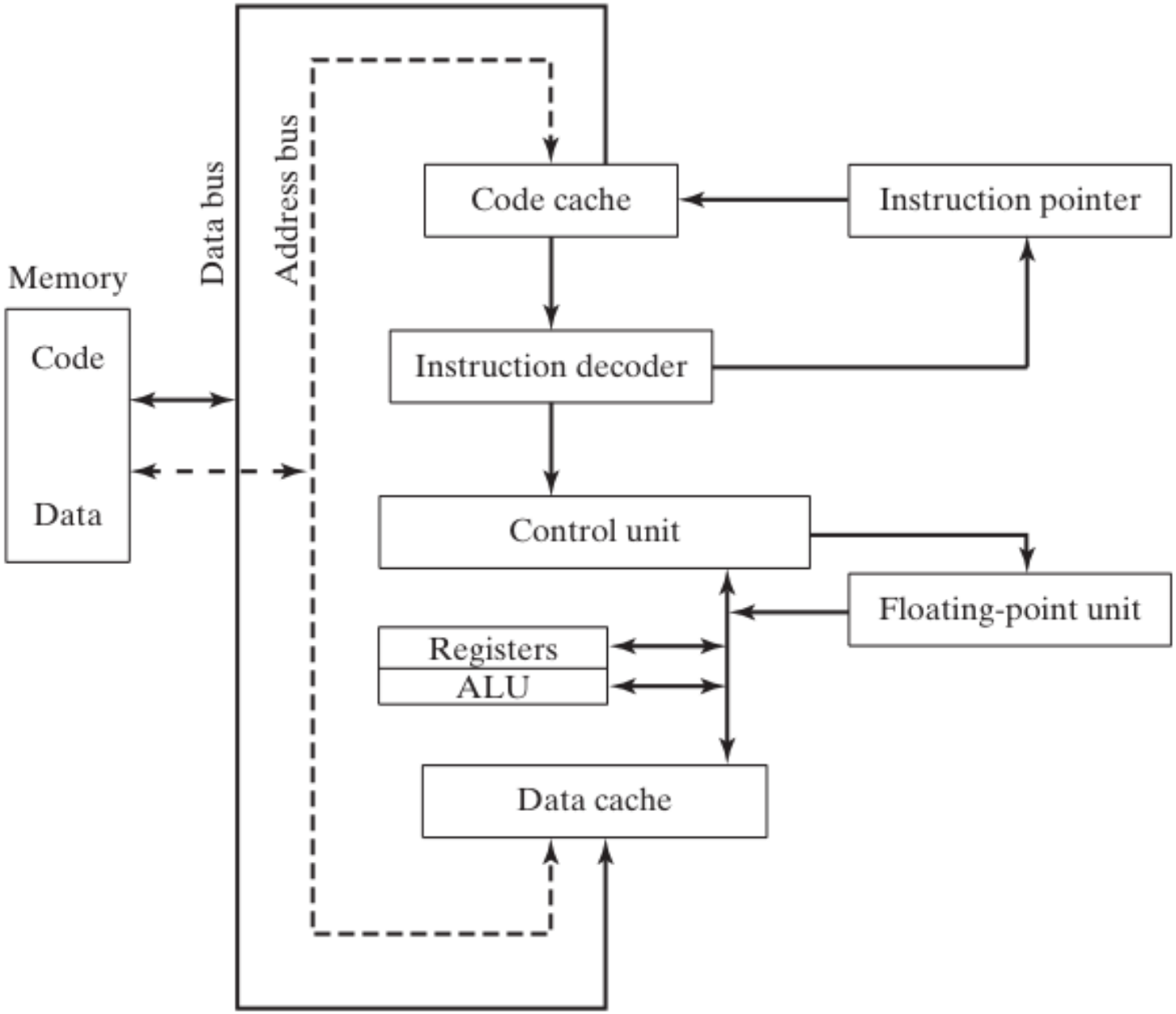
- This sequence of steps is repeated for each instruction executed by the CPU.



FETCH, DECODE, EXECUTE, STORE

The CPU executes instructions by following a sequence of steps: fetching the instruction from memory and incrementing the instruction pointer, decoding the instruction to determine the operation and operands, fetching any necessary operands from registers or memory, executing the instruction and updating status flags, and storing the result if necessary. This process is often simplified to fetch, decode, and execute. The CPU interacts with other components, such as the memory controller, instruction decoder, control unit, and ALU, using digital signals and the system clock. Operand values can be inputs or outputs to an operation, and a block diagram can be used to show data flow and relationships between components during the instruction execution cycle.

FIGURE 2–2 Simplified CPU block diagram.



READING FROM MEMORY

Reading from computer memory is slower than accessing internal registers due to the four steps involved: placing the value you want to read on the address bus, asserting or change the value the processor's RD pin(ReadPin), waiting one clock cycle for the memory to respond, and copying the data from the data bus to the destination operand. To reduce the time spent reading and writing memory, CPU designers developed **cache memory**, which stores recently used instructions and data in high-speed memory. The most commonly used instructions and data are kept in cache memory, which is faster than conventional RAM as it is constructed from a special type of memory chip called **static RAM** that does not need constant refreshing. **Level-1 cache** is stored on the CPU, while **level-2 cache** is attached to the CPU by a high-speed data bus. When data is found in cache memory, it's called a **cache hit**, while a **cache miss** happens when the data is not found in cache.

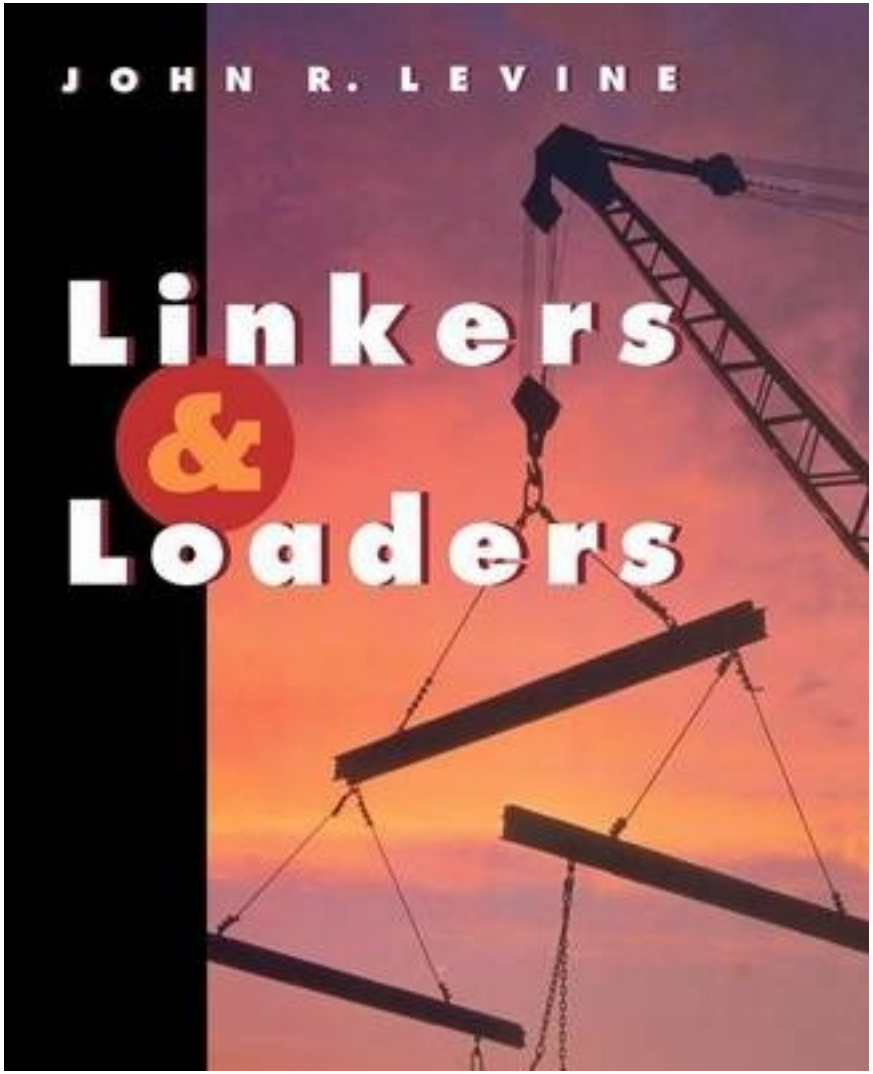
Why does memory access take more machine cycles than register access?

Memory access takes more machine cycles than register access because reading from memory involves multiple steps, including placing the memory address on the address bus, asserting the read signal, waiting for memory to respond, and copying the data from the data bus to the destination operand. Each of these steps typically takes one clock cycle, and the entire process can be slower than accessing registers, which can usually be done in a single cycle. Additionally, memory is generally slower than registers, so even if memory access only took one cycle, it would still be slower than

accessing registers. To address this speed difference, modern CPUs use cache memory to store frequently accessed data and instructions for quicker access.

LOADING AND EXECUTING A PROGRAM.

Before a program can run, it must be loaded into memory by a program loader and the operating system must point the CPU to the program's entry point. The operating system searches for the program's filename and retrieves basic information about the program's file from the disk directory. The OS then loads the program file into memory, allocates a block of memory to the program, and enters information about the program's size and location into a descriptor table. The OS assigns the process an identification number (process ID), tracks its execution, and responds to requests for system resources. When the process ends, it is removed from memory.



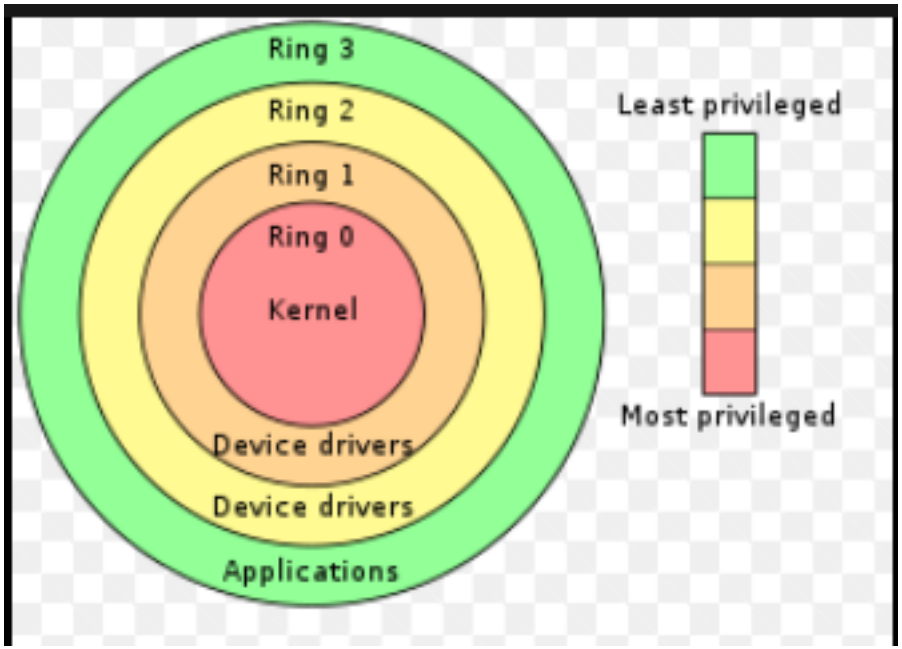
MODES OF OPERATIONS IN X86 PROCESSORS

Here are some bullet points on modes of operation in x86 processors:

x86 processors have different modes of operation, which determine the level of privilege and the types of instructions that can be executed.The modes of operation in x86 processors include real

mode, protected mode, virtual-8086 mode, and long mode.

• **Protected mode** is a more advanced mode that provides memory protection and multitasking capabilities, allowing multiple programs to run simultaneously. Protected mode is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments. Protected Mode, on the other hand, is a more advanced operating mode that was introduced in later x86 processors. It provides a more complex memory model with features such as virtual memory and memory protection. In Protected Mode, kernel ring protection is implemented through hardware support such as privilege levels and protection rings, which restrict access to memory and system resources based on the privilege level of the executing code.



• **Virtual-8086 mode** is a type of protected mode that allows legacy 16-bit applications to run in a protected environment without requiring modifications to the code. While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe environment. In other words, if a program crashes or attempts to write data into the system memory

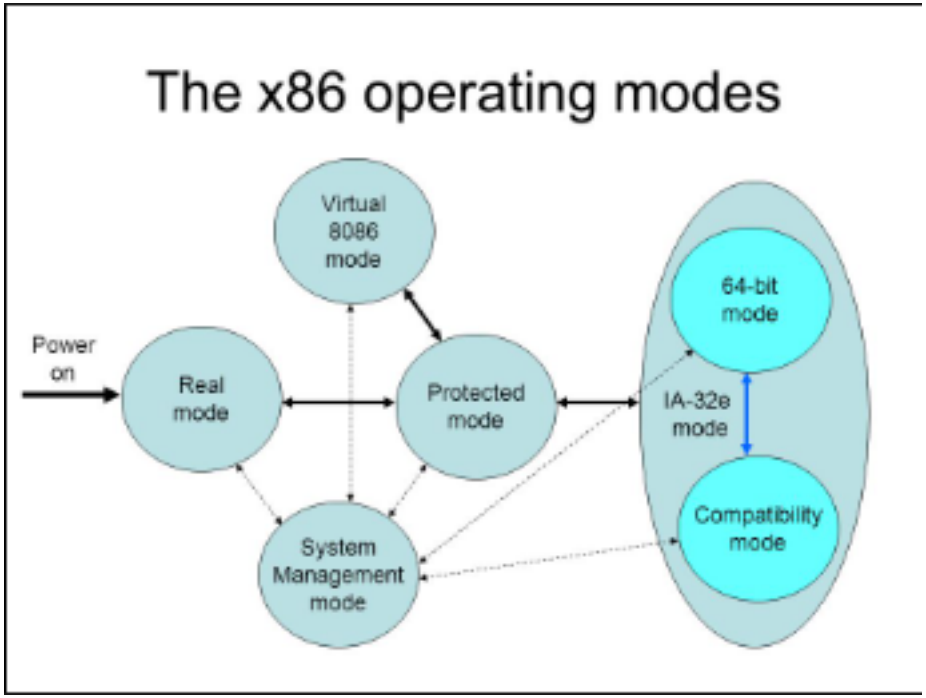
area, it will not affect other programs running at the same time. A modern operating system can execute multiple separate virtual-8086 sessions at the same time.

- **Long mode** is the mode used by 64-bit x86 processors, which provides access to larger amounts of memory and increased processing power.

• **Real-address mode** implements the programming environment of an early Intel processor with a few extra features, such as the ability to switch into other modes. This mode is useful if a program requires direct access to system memory and hardware devices. Real Mode is an operating mode of x86 processors that is compatible with the earliest IBM PC models. It provides a simple memory model and lacks the memory protection features of modern operating systems. In Real Mode, kernel ring protection can be implemented through software techniques such as segmentation, where code and data are separated into different segments and access to these segments is controlled through **segment descriptors.**

• **System management mode (SMM)** provides an operating sys- tem with a mechanism for implementing functions such as power management and system secu- rity. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

To switch between modes in x86 processors, the processor must execute a specific instruction or sequence of instructions, which may require special privileges or access to certain resources.



Each mode of operation in x86 processors has its own set of registers, memory management system, and interrupt handling mechanism.

The operating system typically determines the mode of operation in x86 processors, and applications must conform to the rules and limitations of the selected mode.

ADDRESS SPACE

In 32-bit protected mode, a program or task can access up to 4 gigabytes of memory using linear addressing. This mode was introduced in later x86 processors and provides advanced features such as virtual memory and memory protection. With extended physical addressing, processors starting from the P6 model can address up to 64 gigabytes of physical memory.

On the other hand, real-address mode programs are limited to accessing only 1 megabyte of memory. This mode is compatible with the earliest IBM PC models and lacks the memory protection and virtual memory features of protected mode.

When running multiple programs in virtual-8086 mode within protected mode, each program is allocated its own 1 megabyte memory area. Virtual-8086 mode allows real-mode programs to run in a protected environment, providing them with access to the advanced features of protected mode while maintaining compatibility with legacy software.

In summary, protected mode provides a much larger addressable memory space, advanced features such as virtual memory and memory protection, and compatibility with real-mode programs through virtual-8086 mode. Real-address mode, on the other hand, is limited to accessing only 1 megabyte of memory and lacks these advanced features.

ADDRESSING

Addressing is the process of referring to a specific memory location in a computer's memory system, such as RAM or a hard disk. Every memory location in a computer system has a unique address, which can be used to retrieve or store data from that location.

Address space refers to the range of memory addresses that a particular computer system can access. In other words, it is the total amount of memory that a processor can address or access. The size of the address space is determined by the number of bits used to represent memory addresses.

For example, in a 32-bit computer system, the processor can address a maximum of 4 gigabytes of memory, which is equivalent to 2^{32} individual memory addresses. In contrast, a 64-bit computer system can address a maximum of 18 exabytes of memory, which is equivalent to 2^{64} individual memory addresses.

The size of the address space is an important factor in determining the performance and capabilities

of a computer system. A **larger address space** allows for more memory to be accessed and can improve the performance of memory-intensive applications, while a smaller address space may limit the amount of memory that can be used and restrict the performance of the system.

OPCODE AND OPERAND

In computer programming, an operand is a data value on which an operation is performed, while an opcode (operation code) is a code that represents the operation to be performed by the CPU.

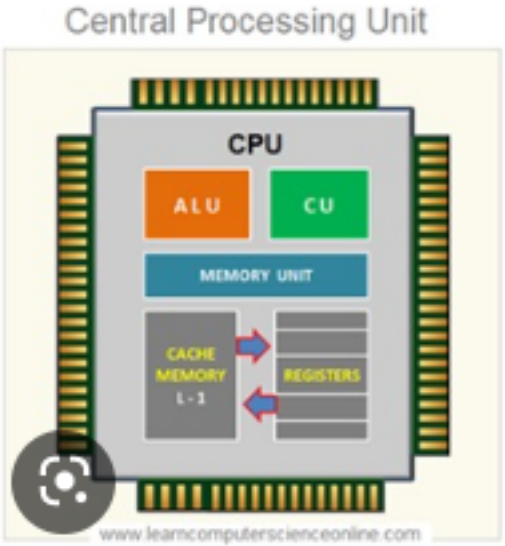
The **opcode** specifies the type of operation that needs to be performed, such as addition, subtraction, multiplication, or division. The **operands**, on the other hand, are the values or variables that the operation is performed on. For example, in the expression "x = y + z", "+" is the opcode, while "y" and "z" are the operands.

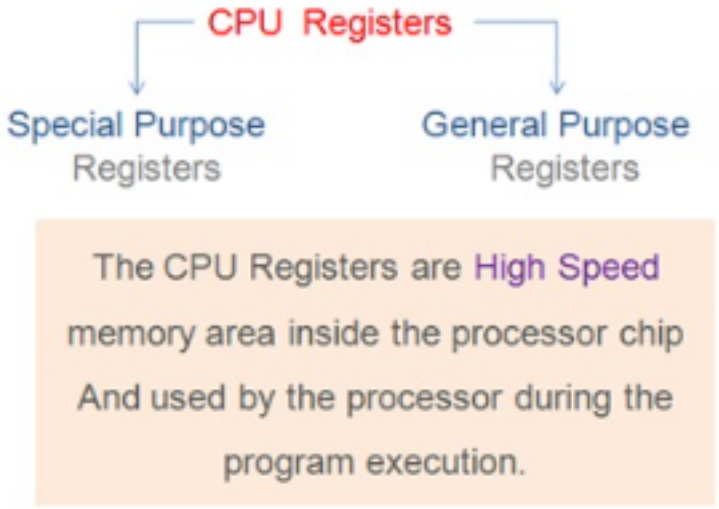
In a CPU's instruction set, an opcode typically appears in the form of a binary code that is associated with a particular operation. The CPU reads the opcode from memory and performs the corresponding operation on the operands that are specified in the instruction.

In summary, an opcode is a code that represents the operation to be performed, while an operand is a data value or variable that the operation is performed on. The **opcode and operands together form an instruction** that is executed by the CPU to perform a specific operation.

GENERAL PURPOSE REGISTERS

Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

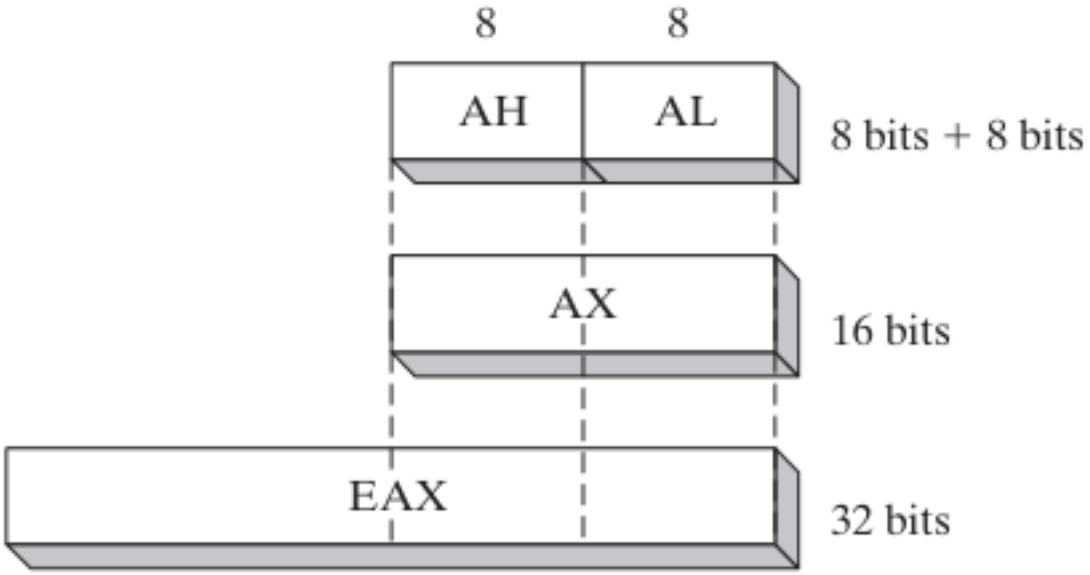




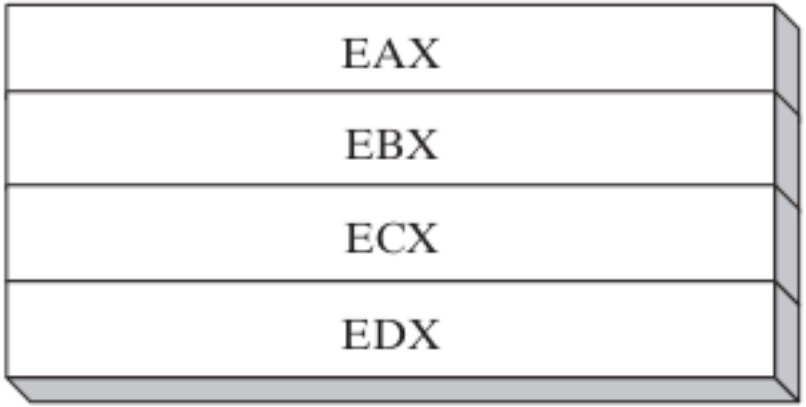
In 32-bit x86 architecture, there are eight general-purpose registers, each 32 bits in size. These registers can be used to hold data, addresses, or pointers, among other things. The eight 32-bit general purpose registers are:

“E” : Extended....

General-purpose registers.



- **EAX (Accumulator Register):** used for arithmetic and logical operations and to store function return values.
- **EBX (Base Register):** used as a base pointer for memory access and to hold data that does not require frequent updates.
- **ECX (Counter Register):** used as a loop counter and for storing small values.
- **EDX (Data Register):** used for I/O operations and for holding high-order results of arithmetic operations.



Each register can hold a 32-bit value. The registers can be modified by arithmetic and logical operations, as well as by memory operations. The registers can be accessed by their full names (e.g., EAX, EBX) or their lower 16-bit or 8-bit parts (AX, AH, AL).

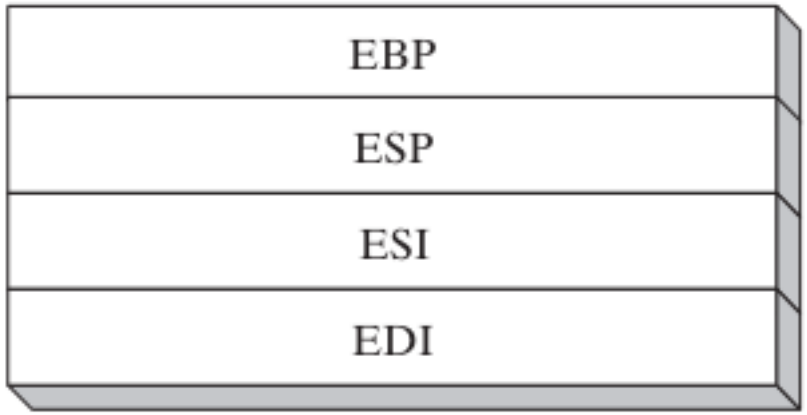
Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL. The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names.

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

- **EBP (Base Pointer Register):** used as a base pointer for stack frames. EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the extended frame pointer register.
- **ESP (Stack Pointer Register):** used to point to the top of the stack. ESP addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the extended stack pointer register. Some registers have specific uses, such as EBP and ESP for stack operations.
- **ESI (Source Index Register):** used as a source pointer for string operations.
- **EDI (Destination Index Register):** used as a destination pointer for string operations. ESI and EDI are used by high-speed memory transfer instructions.

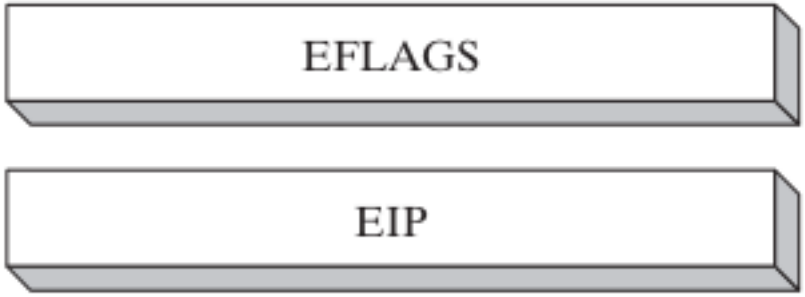


The values in the registers are lost when the program terminates, so they are not persistent.

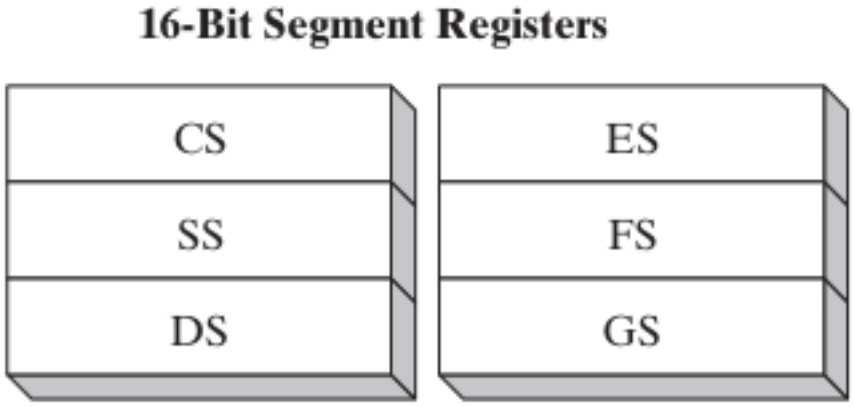
SPECIAL PURPOSE REGISTERS

Here are the names of the special-purpose registers in x86 architecture:

1. Instruction Pointer (IP): points to the memory address of the next instruction to be executed.
2. Flags Register (FLAGS or EFLAGS): contains flags that indicate the state of the CPU after an arithmetic or logical operation.



3. Stack Pointer (SP or ESP): points to the top of the stack.
4. Base Pointer (BP or EBP): used as a base pointer for stack frames. This register is called the base pointer as its most common use is to point to the base of a stack frame during function calls. However, unlike the previously discussed registers, you may use any other register for this purpose if needed.
5. Index Registers (SI, DI, ESI, and EDI): used for string and array operations.
6. Segment Registers (CS, DS, SS, ES, FS, and GS): store the starting address of a memory segment.



7. Control Registers (CR0, CR1, CR2, CR3, CR4, CR8): used to control various aspects of the processor and system, such as memory management and virtualization.
8. Debug Registers (DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7): used for debugging purposes, such as setting breakpoints and monitoring memory access.
9. Model-Specific Registers (MSR): used for processor-specific configuration and performance tuning.
10. Task Register (TR): used for task switching in protected mode.
11. Local Descriptor Table Register (LDTR): points to the local descriptor table in protected mode.

NOTE:

ESP and EBP are both special purpose and general purpose?

Yes, both ESP and EBP are considered special-purpose registers because they have dedicated functions and are not used for general arithmetic or logical operations.

ESP, or the Stack Pointer, is used to point to the top of the stack, which is important for managing program flow and function calls.

EBP, or the Base Pointer, is used as a base pointer for stack frames, which helps with debugging and optimization.

However, ESP and EBP can also be used for general data storage and manipulation, so they can be considered both special-purpose and general-purpose registers. This is in contrast to registers like EAX or EBX, which are used exclusively for general arithmetic and logical operations and are therefore considered general-purpose registers.

"E" in these registers stands for extended, right?

Yes, in x86 architecture, the **"E" prefix** in registers such as EAX, EBX, ECX, and so on stands for "extended." These registers are 32-bit general-purpose registers and are used in **32-bit mode (also known as protected mode)** of the x86 architecture.

The "E" prefix is used to differentiate these 32-bit registers from their 16-bit counterparts, which were used in 16-bit mode (also known as real mode) of the x86 architecture. The 16-bit registers are named AX, BX, CX, and so on.

In **64-bit mode (also known as long mode)** of the x86 architecture, the **"R" prefix** is used for registers that are 64 bits wide, such as RAX, RBX, RCX, and so on.

SEGMENT REGISTERS

Are part of special purpose registers.

Here are some key points about segment registers in x86 architecture:

Segment registers are registers that hold **segment selectors**, which are used to access different memory segments.

In real-address mode, segment registers are 16 bits wide and point to fixed memory areas called segments.

In protected mode, segment registers hold selectors that point to **segment descriptor tables.**

These tables contain **information about the memory segment**, including its base address, size, access rights, and other attributes.

In protected mode, there are six segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment), FS (F Segment), and GS (G Segment).

Each segment has a **base address and a limit,** which are **specified in the segment descriptor table**.

The **base address** is the starting address of the segment, and the limit is the size of the segment. The **limit** is used to prevent programs from accessing memory outside the segment.

- The **Code Segment (CS) register** points to the segment that contains the current code being

executed.

- The **Data Segment (DS) register** points to the segment that contains program data.
- The **Stack Segment (SS) register** points to the segment that contains the program stack.
- The **Extra Segment (ES) register** is a general-purpose segment register that can be used for various purposes, such as holding additional data.
- The **F Segment (FS)** and **G Segment (GS) registers** are additional segment registers that were added in later versions of the x86 architecture. They are typically used for thread-local storage, which allows each thread of a multi-threaded program to have its own copy of certain variables.

In real-address mode, the CPU uses 16-bit segment registers to access memory directly.

These registers include CS (code segment), DS (data segment), SS (stack segment), and ES (extra segment), and they **hold the base addresses** of preassigned memory areas called segments.

In protected mode, segment registers are still used, but they hold pointers to segment descriptor tables (SDTs) instead of directly accessing memory. The SDTs are stored in memory and provide information about the size, location, and access rights of the segments.

In protected mode, there are six segment registers instead of four in real-address mode. The additional registers are FS and GS, which can be used for storing additional data or addressing thread-local storage (TLS).

Each segment descriptor in the SDT contains a base address, a limit, and access rights information, such as **read-only**, **read-write**, **execute-only**, or **execute-read.**

The segment registers are used to calculate the physical memory address that corresponds to a logical address used by a program. This is done by adding the offset of the logical address to the base address of the corresponding segment register.

The segment registers can be **modified** using special instructions, such as MOV, PUSH, and POP. However, **in protected mode**, changing the segment register value will not directly change the physical memory address accessed by a program, since the address is calculated based on the segment descriptor in the SDT.

The values of segment registers can be changed using the "MOV" instruction, which allows a new selector value to be loaded into the register.

The selector is a 16-bit or 32-bit value that points to an entry in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT), which contains information about the segment.

The "MOV" instruction takes two operands: **the destination operand**, which is the segment register that will be modified, and **the source operand**, which is the new selector value to be loaded into the register.

For example, to load a new value into the DS (data segment) register, the following instruction can be used:
MOV **DS**, **new_selector.**

This instruction will load the value of "new_selector" into the DS register, which will then be used to calculate the physical memory address accessed by the program.

It's important to note that changing the value of a segment register does not immediately change the memory segment accessed by the program.

Instead, the processor uses the new selector value to look up the corresponding entry in the GDT or LDT and retrieve the base address and size of the segment.

The memory address accessed by the program is then calculated by **adding the offset of the logical**

address to the base address of the segment.

INSTRUCTION REGISTER

The EIP (Extended Instruction Pointer) is a 32-bit register in x86 architecture that holds the address of the next instruction to be executed by the CPU.

When the CPU fetches an instruction from memory, it reads the opcode from the address pointed to by EIP and then increments EIP to point to the next instruction.

Certain machine instructions, such as JMP (jump) and CALL (call subroutine), can manipulate the value of EIP to cause the program to branch to a new location in memory.

For example, a JMP instruction can be used to unconditionally jump to a different location in the code, while a CALL instruction can be used to call a subroutine and save the return address in the EIP register.

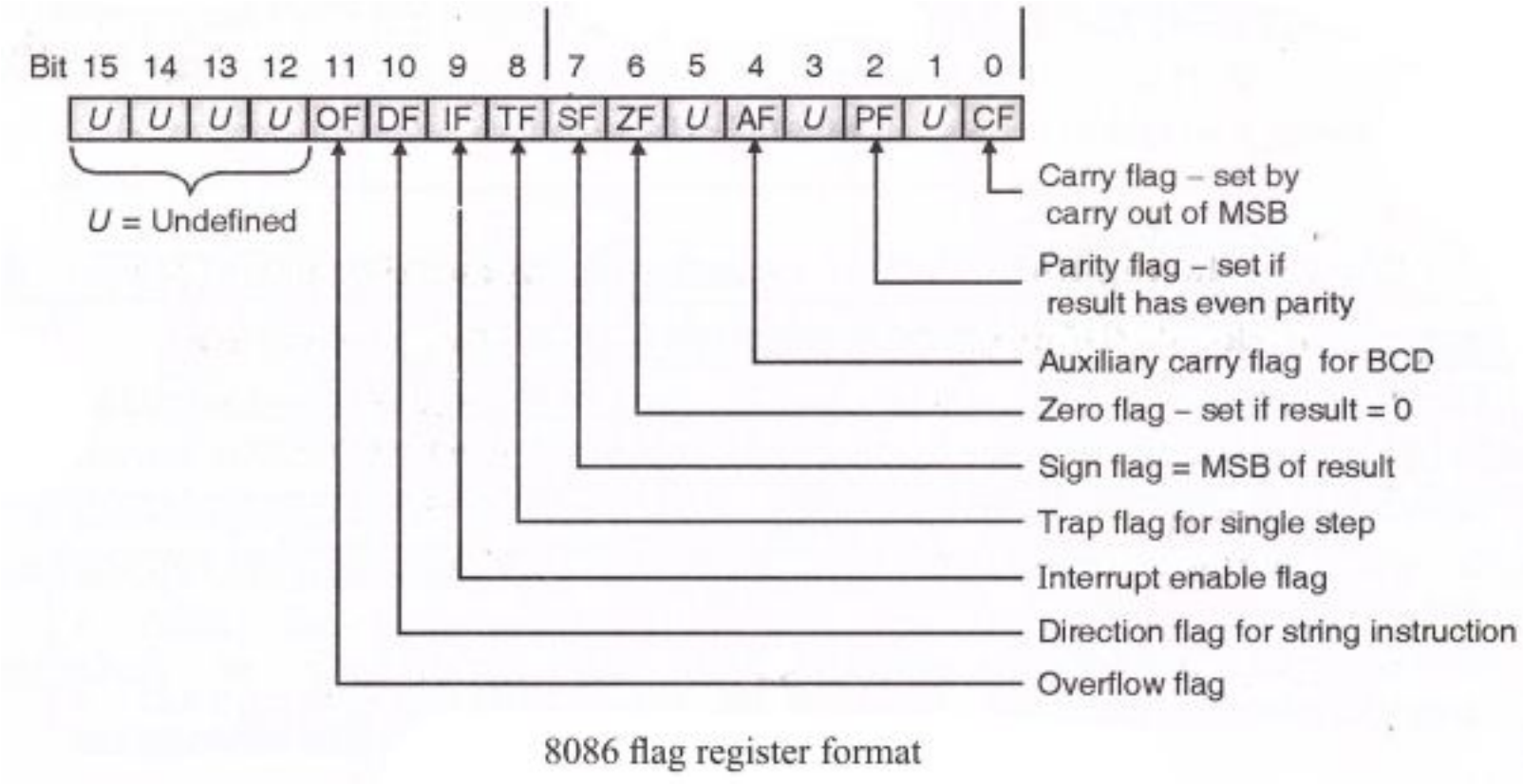
The EIP register is part of the larger EFLAGS (Extended Flags) register, which contains various CPU flags and control bits that are used to control the behavior of the CPU and indicate the results of arithmetic and logical operations.

The EIP register can be modified using certain machine instructions, such as JMP, CALL, RET (return from subroutine), and INT (interrupt), as well as by software interrupts or hardware interrupts that cause the CPU to jump to a different location in memory.

FLAG REGISTERS

The **EFLAGS or FLAGS register** is a 32-bit register that contains various control bits and flags that are used to control the behavior of the CPU and indicate the results of arithmetic and logical operations. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode. Programs can set individual bits in the EFLAGS register to control the CPU’s operation. Examples are the **Direction** and **Interrupt** flags.

"Control Flag" is not a specific flag in the **x86 EFLAGS register.** However, the EFLAGS register **as a whole can be thought of as a control register**, since its various flags and control bits are used to control the behavior of the CPU and its operations. The EFLAGS register is sometimes referred to as the **"Flag Register"** or the **"Status Register"**, but these terms are not always used consistently across different sources or programming environments. In general, it's important to consult the specific documentation or reference material for a given software or hardware platform to understand the meaning and usage of specific flags and registers.



8086 flag register format

Carry Flag (CF): This flag is set when an arithmetic operation generates a carry or borrow out of the most significant bit of the result, and is used to support multi-word arithmetic operations and binary-coded decimal (BCD) arithmetic.

Parity Flag (PF): This flag is set if the least significant byte of an operation result contains an even number of set bits (i.e., the result has even parity), and is used for parity checking and error detection.

Auxiliary Carry Flag (AF): This flag is set when an arithmetic operation generates a carry or borrow out of the low nibble (bits 0-3) of the result, and is used to support BCD arithmetic.

Zero Flag (ZF): This flag is set when an operation generates a result of zero, and is used to test for null pointers and empty data structures.

Sign Flag (SF): This flag is set when the most significant bit of an operation result is set (i.e., the result is negative), and is used for signed arithmetic and conditional branching.

Trap Flag (TF): The trap flag (TF) is a specific flag in the x86 EFLAGS register that is used to enable single-step debugging. When the TF is set (i.e., 1), the CPU will automatically generate a trap interrupt after executing each instruction, allowing a debugger to inspect the state of the program and step through the code one instruction at a time. The trap flag is typically used in conjunction with hardware or software breakpoints, which temporarily modify the instruction stream of a program in order to halt its execution at a specific point. By setting the TF flag, a programmer or debugger can control the flow of a program and observe its behavior in detail, making it easier to diagnose and fix bugs and other issues. It's worth noting that the use of the TF flag can slow down the execution of a program considerably, as it adds additional overhead to each instruction cycle. For this reason, it is typically **used only when debugging or profiling code**, and is disabled during normal program execution.

The Direction flag (DF) controls the direction of string operations (such as REP MOVSB, REP STOSB, etc.) that copy or move blocks of data between memory locations. When the DF is set (i.e., 1), the string operations proceed in reverse order, from high memory addresses to low memory addresses. When the DF is clear (i.e., 0), the string operations proceed in forward order, from low memory addresses to high memory addresses.

The Interrupt flag (IF) controls the ability of the CPU to respond to external interrupts from devices or other sources. When the IF is set (i.e., 1), the CPU can respond to interrupts by executing the appropriate interrupt service routine (ISR) in response to an interrupt request. When the IF is clear (i.e., 0), the CPU disables external interrupts and will not respond to them.

Overflow Flag (OF): This flag is set when an arithmetic operation generates a result that is too large or too small to be represented in the available bit width of the operands, and is used to

detect signed arithmetic overflow and underflow.

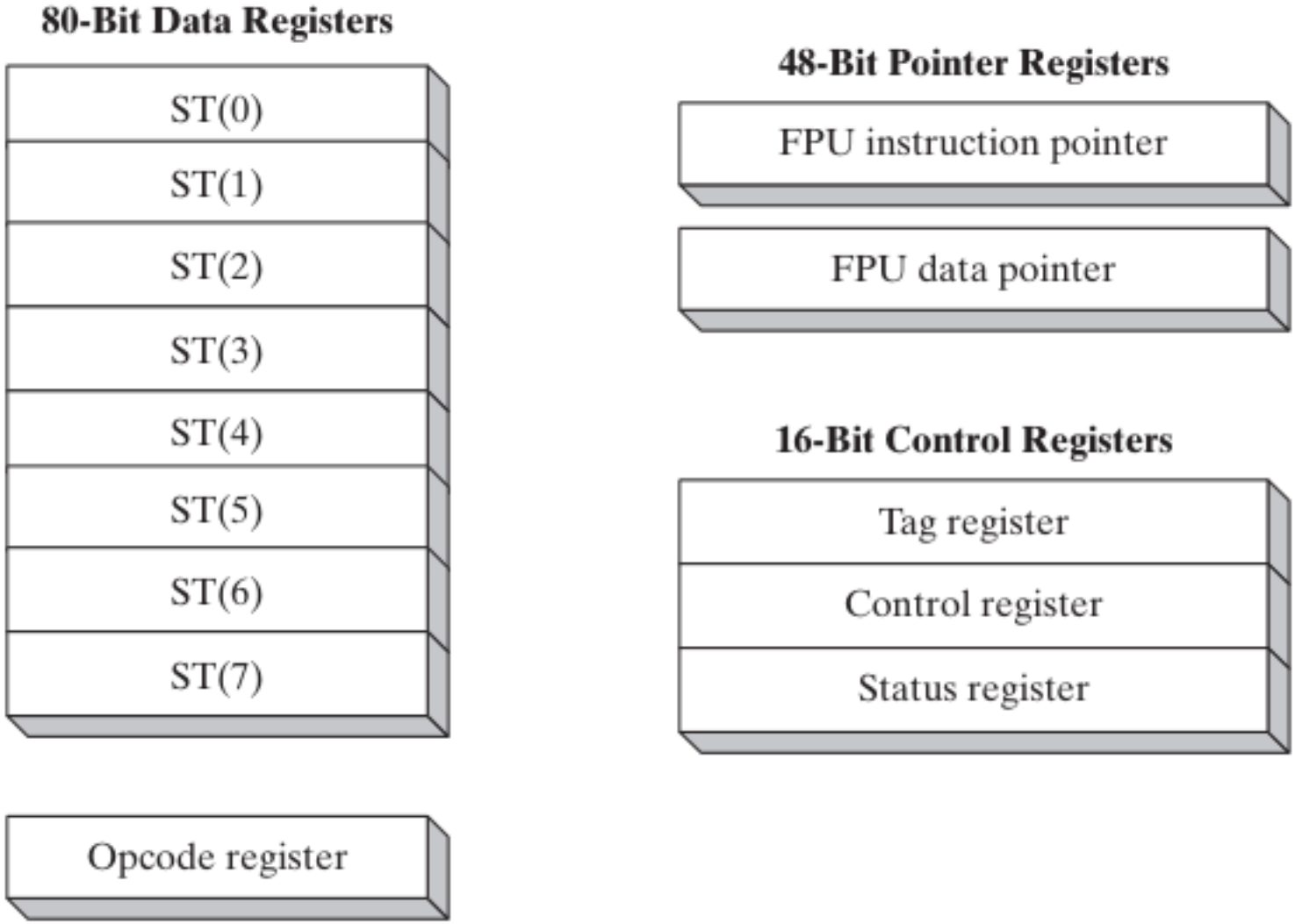
MMX AND XMM REGISTERS

The **MMX technology** is a feature present in Intel processors that significantly improves their performance when implementing advanced multimedia and communication applications. These processors come equipped with eight 64-bit MMX registers that support **SIMD (Single-Instruction, Multiple-Data) instructions.** These instructions operate in parallel on data values contained in the MMX registers. While these registers appear to be separate, they are actually aliases for the same registers used by the floating-point unit.

The x86 architecture also includes eight 128-bit registers called **XMM registers,** which are used by streaming SIMD extensions to the instruction set. These registers are designed to enhance the performance of multimedia and signal processing applications. The MMX and XMM registers allow for the efficient handling of multimedia and other data-intensive tasks, providing developers with tools to optimize their software for Intel processors.

The **floating-point unit (FPU)** is responsible for high-speed floating-point arithmetic. In the past, a separate coprocessor chip was required to perform these calculations. However, from the Intel486 onwards, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, each named ST(0) through ST(7). These registers allow for the efficient processing of floating-point numbers and other **complex mathematical operations.**

FIGURE 2–5 Floating-point unit registers.



Overall, the MMX, XMM, and FPU registers provide Intel processors with the ability to perform complex mathematical calculations and multimedia processing at high speeds. They offer developers a range of tools to optimize their applications for Intel processors, ensuring maximum performance and efficiency.

Register Name	Size	Description
FPU Data Registers	80-bit	There are eight FPU data registers named ST(0)-ST(7). These registers are used to store floating-point numbers and other complex mathematical operations.
Opcode Register	16-bit	The opcode register contains the opcode for the currently executing instruction. This register is read-only and is updated by the processor during instruction execution.
48-bit Pointer Regs.	48-bit	The 48-bit pointer registers are used in protected mode to hold pointers to segment descriptor tables. These registers are used to calculate the physical address of memory locations in protected mode.
Control Registers	16-bit	There are four control registers named CR0-CR3. These registers contain system control flags that govern the behavior of the processor. They are used to set up memory management, protection, and other system-level features. Control registers are read-only in user mode and read/write in supervisor mode.