**8086/8088 Microprocessor Internal Architecture**

Table of Contents

[Internal Architecture 3](#_Toc83226747)

[Pin Diagram 6](#_Toc83226748)

[Registers 7](#_Toc83226749)

[General Purpose Registers 7](#_Toc83226750)

[Memory Segments and Offsets 8](#_Toc83226751)

[Dynamic RAM 10](#_Toc83226752)

[Overlapping Segments 10](#_Toc83226753)

[Segment Registers 12](#_Toc83226754)

[Instruction Pointer and Code Segment Register 13](#_Toc83226755)

[Data Segments and Index Registers 14](#_Toc83226756)

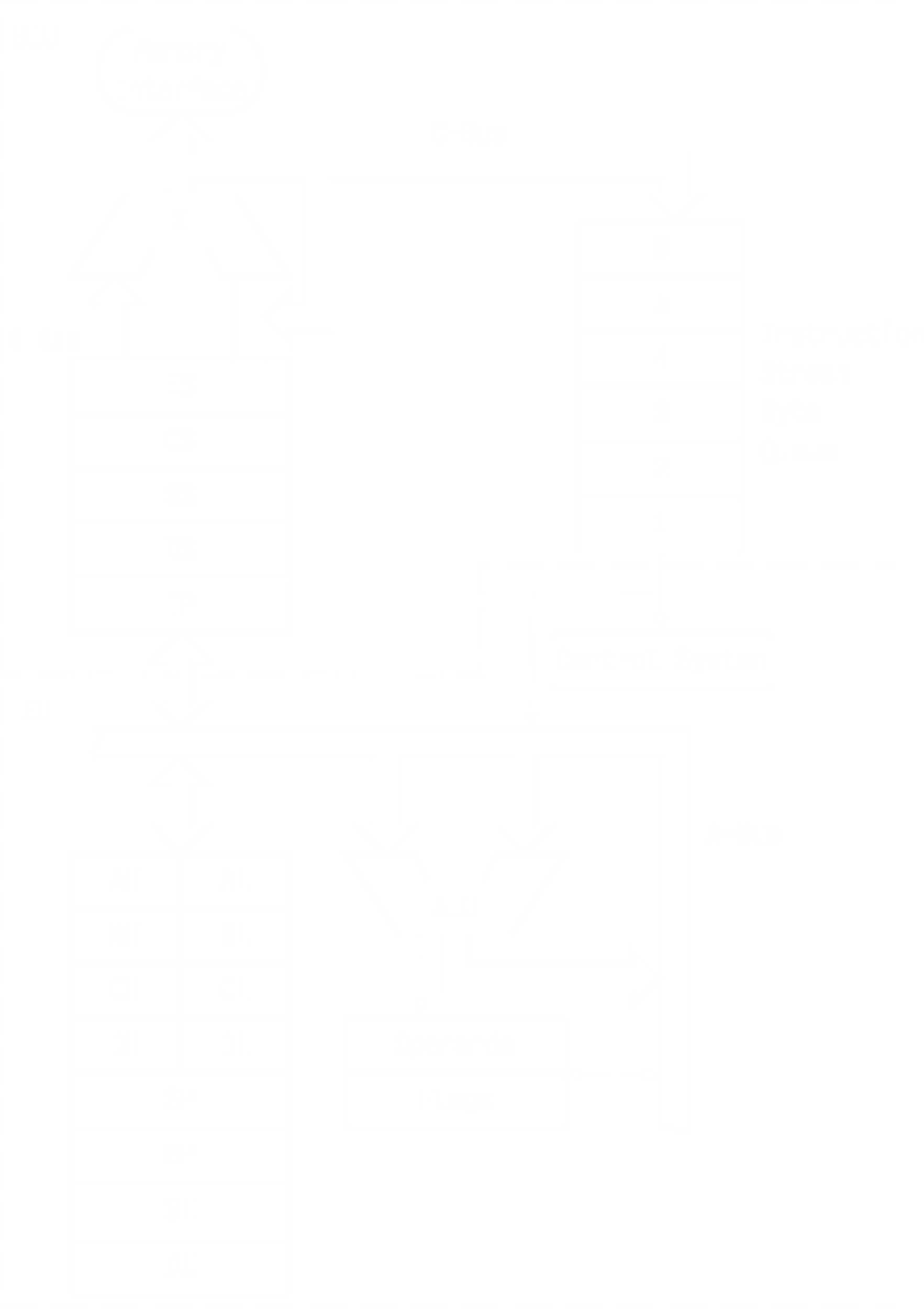
[Stack Pointer and Index Registers 14](#_Toc83226757)

[Flag Registers 15](#_Toc83226758)

The Intel 8086 and 8088 microprocessors have the same **internal architecture** and **instruction set**, which is why we are studying them together. They both have a **20-bit address bus**, but the difference is that the Intel 8086 has a **16-bit data bus**, while the Intel 8088 has an **8-bit data bus**. This difference causes the Intel 8088 to be **cheaper**. This is comparable to the how modern microprocessors can have a 32-bit data bus or a 64-bit data bus. Further versions of the Intel 8086 microprocessors include the Intel 80286, Intel 80486 and Intel Pentium.

## Internal Architecture

The **internal architecture** for the two microprocessors is shown below:



While examining any microprocessor, the first things we need to pay attention to are the **ALU**, the **CU** and the **registers**. The ALU and the CU are clearly marked in this diagram, and everything else (other than the buses of course) are registers.

This diagram is divided into two sections, the **execution unit** (EU), which deals with actual actions the microprocessor takes, and the **bus interface unit** (BIU), which facilities communication between the microprocessor and external interfaces like memory and I/O devices. The BIU is responsible for transmitting address, data and control signals on the buses.

In the **execution unit**, we have the **CU**, the **ALU** and the most important **registers**. The registers include:

1. Four general purpose registers, AX, BX, CX and DX, which are all logically divided into two parts, high and low.
2. Stack Pointer, SP
3. Base Pointer, BP
4. Source Index Register, SI
5. Destination Index Register, DI
6. Flags Register

The SP, BP, SI and DI registers hold addresses to specific memory locations.

In the **bus interface unit**, we have the **instruction pointer** register and four **segment registers**:

1. Extra Segment Register, ES
2. Code Segment Register, CS
3. Stack Segment Register, SS
4. Data Segment Register, DS

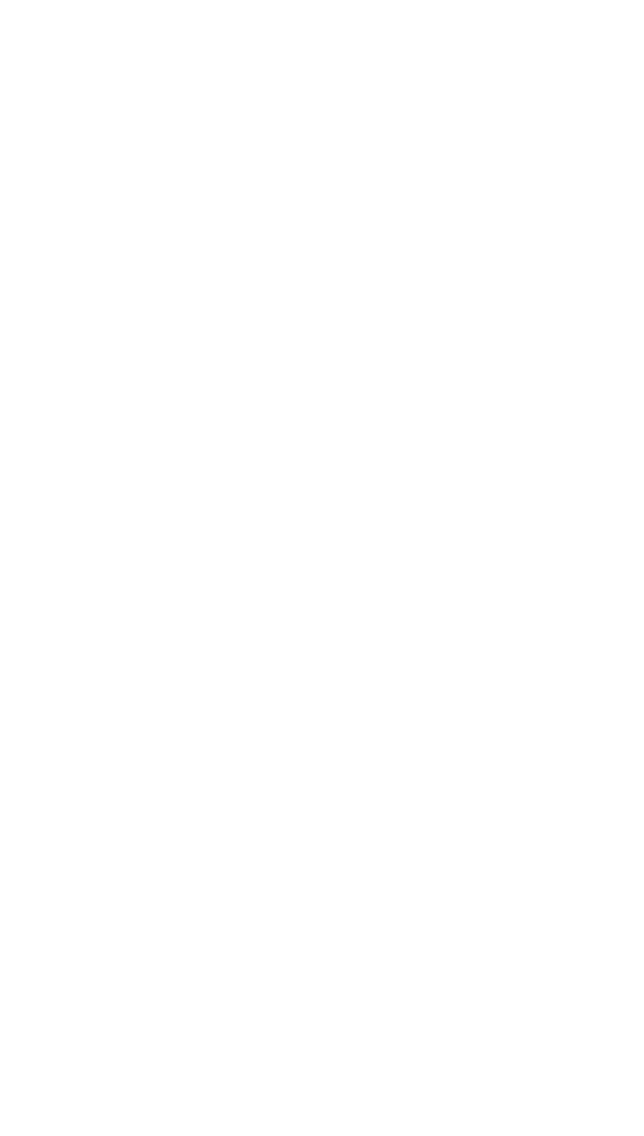
The segment registers hold addresses of memory locations.

Therefore, there are a total of **14 registers**, each of 16-bits, even the flag register, which uses 9 bits.

There are also three **buses**, the A-Bus, the B-Bus and the C-Bus. The reason they are named in this way and not identified as the Data Bus, the Control Bus or the Address Bus is because all three buses can carry data, address or control signals.

Another difference between this microprocessor and the Intel 8085 microprocessor is the lack of the **instruction register**. Instead, we have an **instruction stream byte queue**, which is a buffer with **six slots**, each of which can store 2 bytes, or a **word**.

## Pin Diagram



The Intel 8086 has **40 pins** in DIP. Details of the different pins will be discussed later.

## Registers

The 14 registers in the Intel 8086 can be divided into 6 categories:

* 4 **general purpose** registers, AX, BX, CX and DX
* 2 **pointer** registers, SP and BP
* 2 **index** registers, SI and DI
* 4 **segment** registers, CS, DS, SS, and ES
* 1 **instruction** register, IP
* 1 **flag** register, FR

### General Purpose Registers

The general-purpose registers are meant to hold **data** as opposed to addresses.

AX is the **accumulator**. It is the most efficient register for **arithmetic** and **logic** operations as well as **data transfer**. Using the AX register for these purposes results in the shortest code. This is because, for multiplication and division operations specifically, it is not possible to perform the operations without using the AL or the AX register.

BX is the **base address** register. It is meant to hold addresses to be used for **indirect addressing**.

MOV AX, [1234h] *; direct addressing*MOV BX, 1234h  
MOV AX, [BX] *; indirect addressing*

ASSEMBLY

CX is meant to be used as a **counter** for looping operations as well as shift or rotate operations.

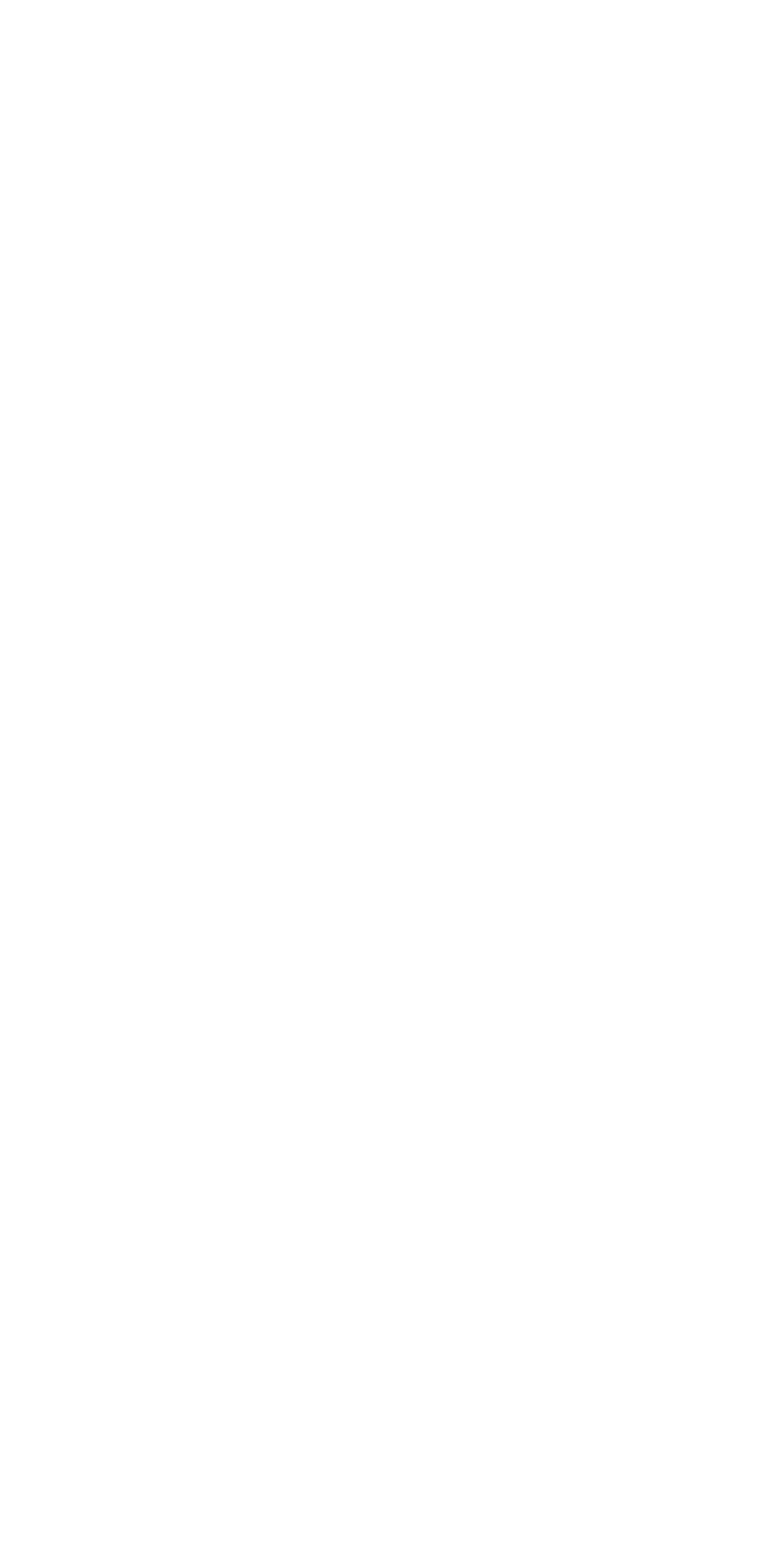
DX is meant to hold **data** to be used in multiplication or division operations. It is also used for I/O operations.

### Memory Segments and Offsets

The rest of the registers all deal with **memory**. Before moving on to them, we need to understand how memory is handled in the Intel 8086 microprocessor.

Since the microprocessor has a **20-bit address bus**, there are locations. Since each location holds 1 byte, this results in **1 megabyte** of memory. Representing 20 bits requires **5 hex digits**, from 00000 to FFFFF.

However, the registers are all of **16 bits**, which suggests that the registers will be unable to store the memory addresses. To get around this issue, the memory is partitioned into **segments**. The entire memory is divided into 16 segments, meaning all the memory locations in each segment can be identified using 16 bits, or **4 hex digits**, which holds 64 kilobytes.



Each segment is identified by a **segment number**, which is the first digit of the original address, followed by 3 zeroes. Within a segment, a specific memory location is identified by an **offset number**. Generally, the segment number is held in the **segment registers**, CS, DS, SS and ES, and the offset number is stored in the **offset registers**, SP, BP, SI and DI. Thus, we require **two registers** to hold a single address.

The segment number defines the starting point of the segment within the memory space, while the offset number is the distance from the beginning to a particular location within the segment.

For example, for the memory location 80F10, the segment number is 8000 and the offset number is 0F10. This is represented as 8000:0F10. We add a 0 to the end of the segment number and add it with the offset. Once the actual address is calculated, it is not stored anywhere (it would not be possible to store it anyways), but instead, passes immediately to the address bus.

## Dynamic RAM

We have seen how the total RAM is divided into 16 segments, each of 64 KB. A single program may need more than 64KB or less. Since we do not know this before we actually load the program, we can face one of two problems. We could either run out of memory while loading the program, or we could end up wasting memory by allocating too much. The second problem essentially occurs if we allocate one segment to each program, which is what happens with **static RAM**. This also means that we would only be able to load 16 programs onto our RAM at once, since we only have 16 segments.

To get around these issues, we use **dynamic memory**.

### Overlapping Segments

A single segment is supposed to be of 64KB. The first segment is from 00000 to 0FFFF. If we load a small program that only takes up 10 memory locations, the program’s memory usage is from 00000 to 0000F. However, we have allocated it the entire segment, so the rest of the segment is wasted.

In dynamic memory, the next program would be loaded into the **immediate next memory location** available, 00010. That program is still given its own segment though. In reality, the next segment itself begins at that location. Thus, several locations seem to belong to both the first and second segments. The segments **overlap**.

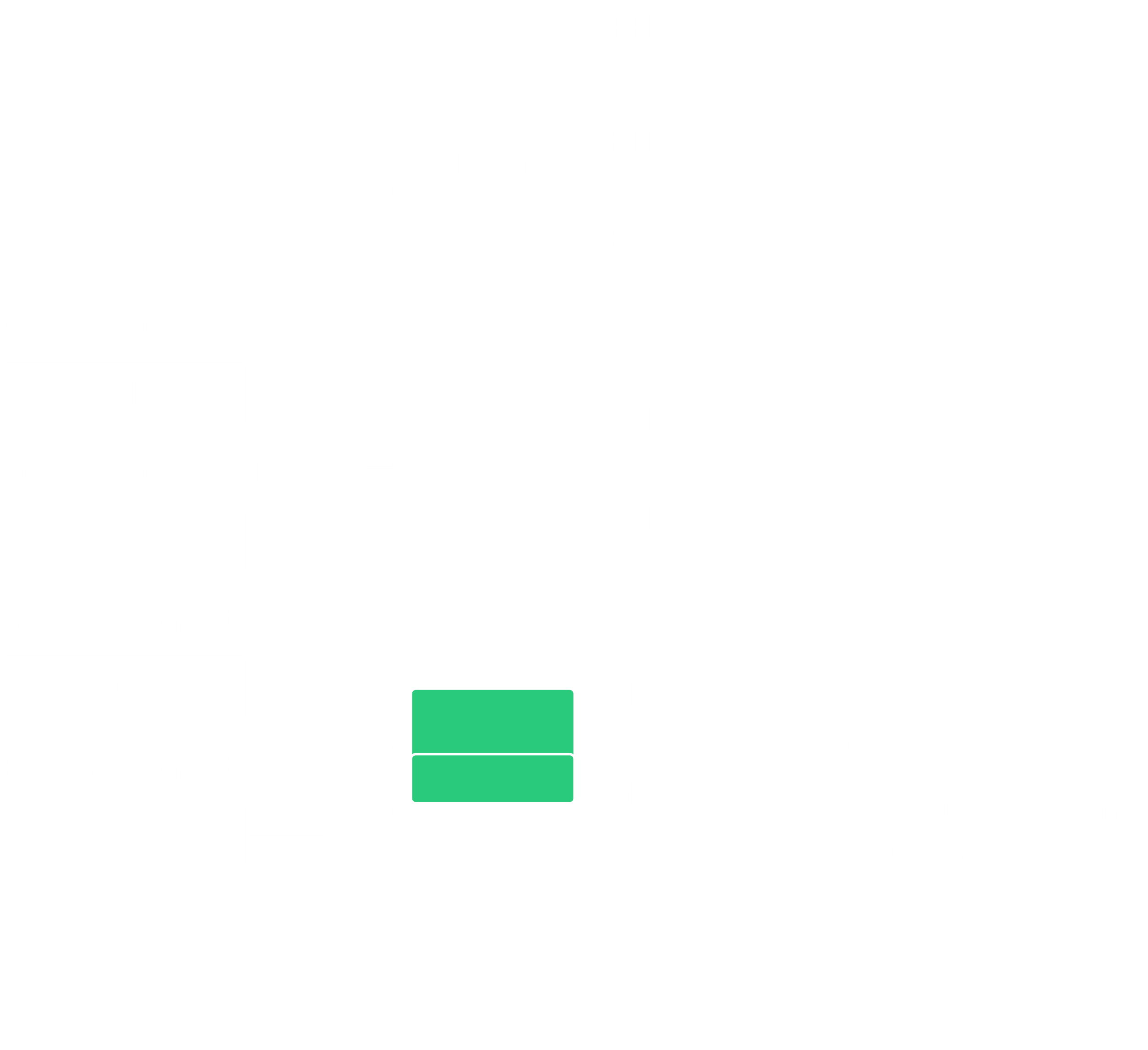
Visually, it looks like this:



Overlapping segments in this way ensures that **memory utilization** is higher.

### Segment Registers

* **Code Segment** (CS) – This holds the segment number of the segment at which the program code begins.
* **Data Segment** (DS) – This holds the segment number of the segment at which the program data begins.
* **Extra Segment** (ES) – This holds an alternative segment number for the program data or code, in case the original one is insufficient.
* **Stack Segment** (SS) – This holds the segment number at which the stack begins, which is used for sub-program executions.



The image above shows how the different segments are distributed in a sample program. These four segments are allocated for every program.

There is a **code segment**, which holds the different instructions of the current program along with any parameters. There is a **data segment**, which holds the values of different variables to be used by the program. Each of these segments are 64KB each.

The **extra segment** holds some extra data or code that did not fit into the 64KB allocated to the data or code segment. The **stack segment** holds data for anything that is pushed onto the stack.

Notice that the stack segment and the extra segment **overlap**, since the stack segment does not need the entire 64KB allocated to it.

### Instruction Pointer and Code Segment Register

We know that the **instruction pointer** holds the address of the next instruction to be executed. In reality, this address is the **offset** relative to the **code segment** as indicated by the code segment register.

For example, if the code segment register has a value of B3FFh and the instruction pointer has a value of 12ABh, the actual instruction is located at the memory location B3FF0h 12ABh B529Bh.

### Data Segments and Index Registers

Consider that we want to store some string ‘ABCDEF’. This will be stored in a **data segment**. Say the data segment has the segment number E000h. The **source index** (SI) register will hold the offset number relative to the DS which can be used to figure out where the string starts. The **destination index** (DI) will hold the offset number relative to the DS which can be used to figure out where the string ends.

This also allows us to perform string operations. For example, if we want to know the location of the fifth alphabet in the string, we can add 4 to the SI (since the SI indicates the first alphabet).

### Stack Pointer and Index Registers

In the 8085 microprocessors, we saw that the stack pointer started at the **top** of the stack instead of the bottom. Each push operation caused the pointer to go down by one memory location, since the memory locations could each hold 8 bits and the 8085 microprocessors were 8-bit microprocessors.

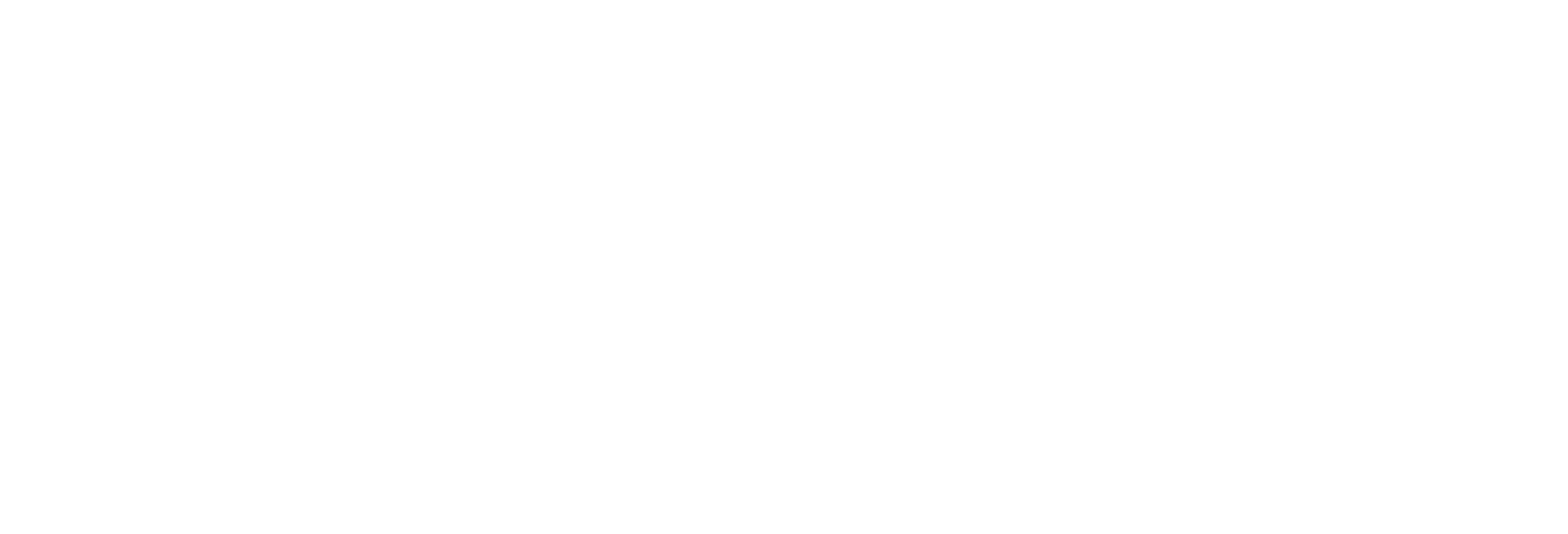
The 8086 microprocessors on the other hand, are 16-bit microprocessors. The stack pointer needs to indicate two memory locations because of this.

Say the **stack segment** (SS) register has the stack number 5D27h. Within the segment, the topmost offset number is FFFFh. However, the **stack pointer** (SP) will initially hold the offset number FFFEh, to indicate that both offset locations FFFEh and FFFFh are occupied by the stack pointer. If we perform a push operation, the stack pointer will move down two locations and the offset number will become FFFCh.

We also have the **base pointer** (BP), which behaves similarly to the SP. The BP can be used to point to any element in the stack.

## Flag Registers

The **flag register** is a 16-bit register that specifies the status of the CPU and information about the results of arithmetic operations. The flag register is **modified automatically** by the CPU after mathematical operations. This allows us to determine the type of the result.



In the diagram above, we can see that the 8086 microprocessor has 5 flags in common with the 8085 microprocessor - the **Sign** flag, the **Zero** flag, the **Auxiliary Carry** flag, the **Parity** flag and the **Carry** flag. These 5 flags behave in the exact same way as they did in the 8085 microprocessors.

However, we have 4 extra flags in the 8086 microprocessors.

1. The **Trap** flag (TF) is used for **on-chip single-step debugging**. When we want to debug a program in single-step mode, this flag is set to . This causes the program to stop after every line of code and wait for the user.
2. The **Interrupt Enable** flag (IF) informs other ICs about the status of interrupts. Other ICs can send **interrupt requests** through the Interrupt Request (INTR) pin or the Non-Maskable Interrupt (NMI) pin. However, they will only send the request if the IF is set to . If it is set to , it means the microprocessor is not accepting interrupts right now and they do not send any requests.
3. The **Direction** flag (DF) is used when processing data chains. Normally, instructions are executed in the ‘forward’ direction. However, it is possible to set a **label** somewhere in the code and use a **jump statement** to get to that point of the code from anywhere else in the program. This is similar to the goto statement from C. This makes it possible to go ‘backwards’ in the code. DF is set to if we are going forwards and set to if we are going backwards.
4. The **Overflow** flag (OF) indicates an overflow in a **signed operation**. For example, if we have a signed number, 01111111, the first indicates the sign, meaning this is a positive number. If we add 1 to this, we will get 10000000. However, the 1 here indicates that this is , which cannot be true, since we performed an addition. This causes the microprocessor to figure out that an overflow has occurred and it sets OF to .

OF may seem similar to CF, but there is an important difference. CF is set to if the overflow is carried outside of the register’s bounds. For OF, we can still see the carry, since we are using signed numbers.

Notice that we are only using 9 of the 16 bits in the flag register, and these bit positions aren’t consecutive. The remaining 7 bit positions are unused or **undefined**.

Example

Consider the same example as before, that we store 7Fh (01111111b) in AL and then add 1. This causes the value of AL to become 80h (10000000b).

In this case, the flag register will look like this:

* CF is , since there is no carry outside of AL’s capacity
* PF is , since 80h has an odd number of s
* AF is , since there is carry out of the 4th bit (out of bit 3 into bit 4)
* ZF is , since the result is not
* SF is , since the sign bit is
* OF is , due the reasons specified above

Since we do not know anything about how the system itself is currently operating, we cannot define TF, IF and DF. We could have defined them if we had the required information.