**8085 Microprocessor Internal Architectures**

Table of Contents

[Block Diagram 3](#_Toc83226708)

[Pin Diagram 5](#_Toc83226709)

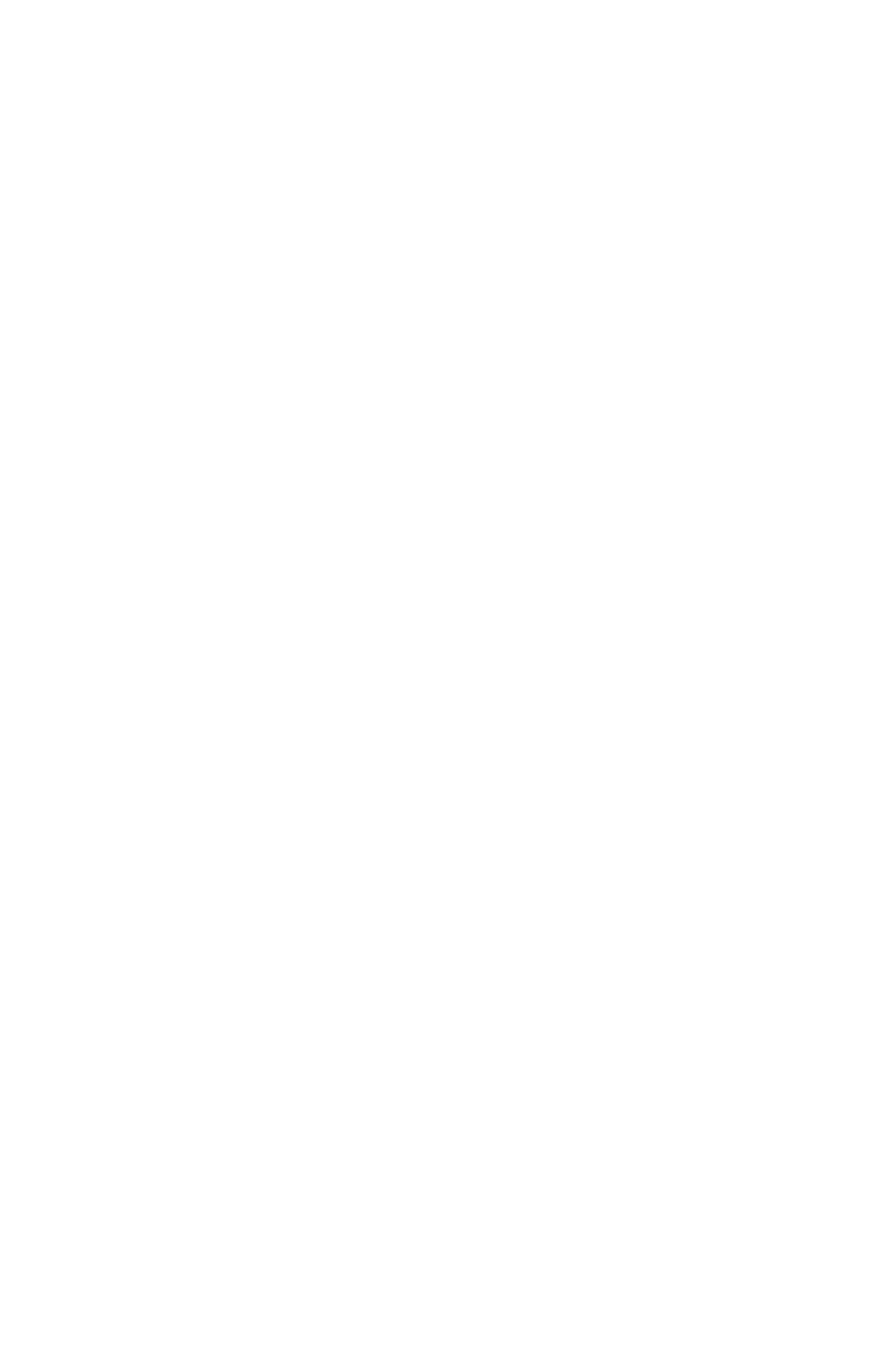
[Memory 7](#_Toc83226710)

[Indirect Addressing 8](#_Toc83226711)

[Flag Register 8](#_Toc83226712)

[Stack Pointer and Stack Memory 10](#_Toc83226713)

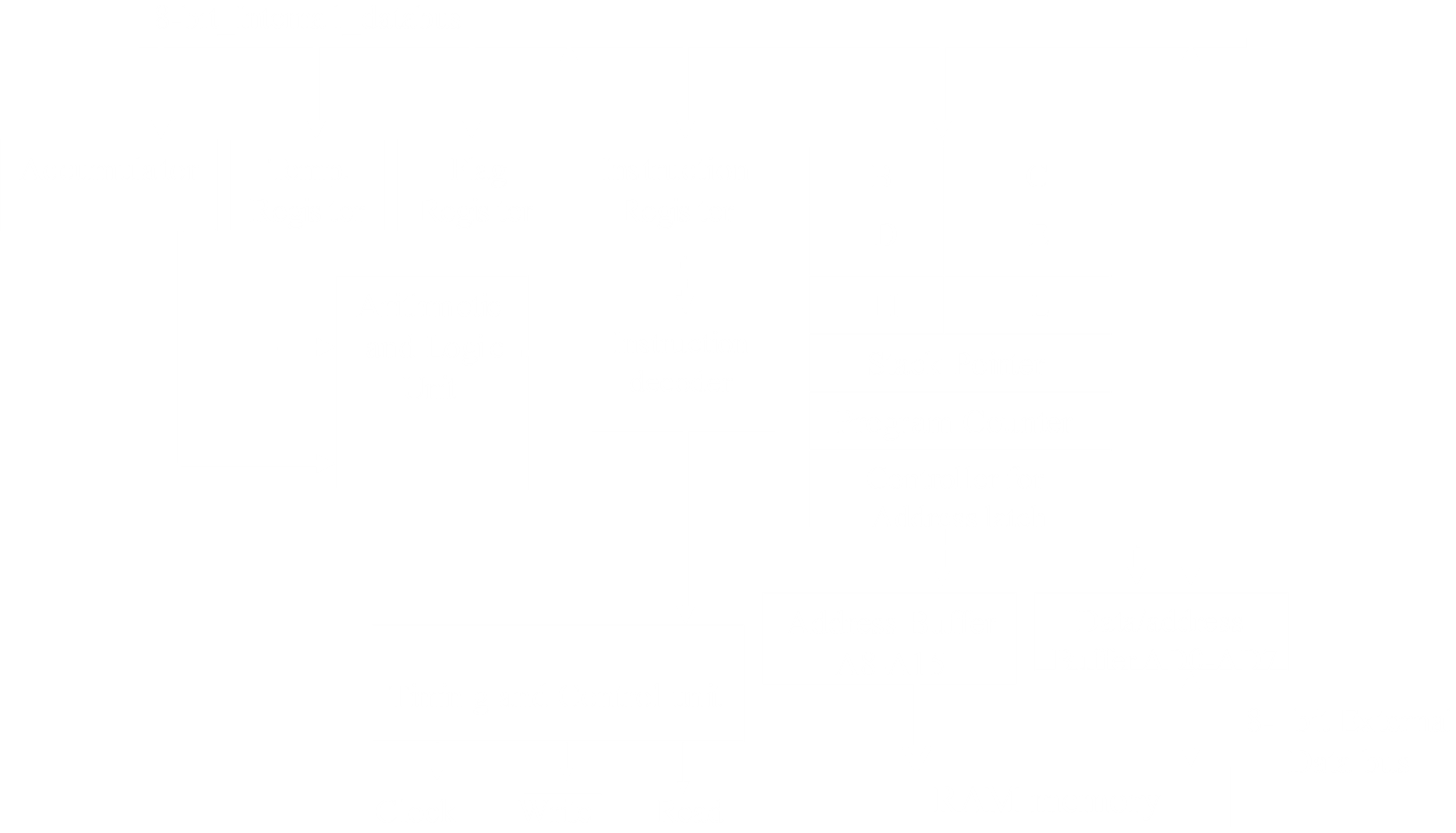
The Intel 8085 is an **8-bit** microprocessor with a **16-bit address bus** and an **8-bit data bus**. It has **40 pins**, in a Dual in Line Package (**DIP**), meaning it has 20 pins each on two sides vertically, as shown in the **Pin Diagram** below.



In Assembly Language, the Intel 8085 had **74 operation codes** which could be used to generate a total of **246 instructions**. For example, consider the instruction MOV. This instruction has a particular op-code. However, the instruction could be executed in a few ways, depending on what we want to achieve. MOV A, B could be one instruction, while MOV C, D is another. Thus, with the same op-code, we have achieved two different instructions.

## Block Diagram

The **Block Diagram** for the Intel 8085 is given below:



There are **three** basic parts to a microprocessor:

1. ALU
2. Registers
3. Control Unit

The **ALU** and **CU** are clearly marked in the diagram above, and additionally, we have several **registers**.

Firstly, we have several **8-bit** registers:

1. Accumulator
2. Temporary Register
3. Flag Register
4. Instruction Register
5. B Register
6. C Register
7. D Register
8. E Register
9. H Register
10. L Register

There are also two **16-bit** registers:

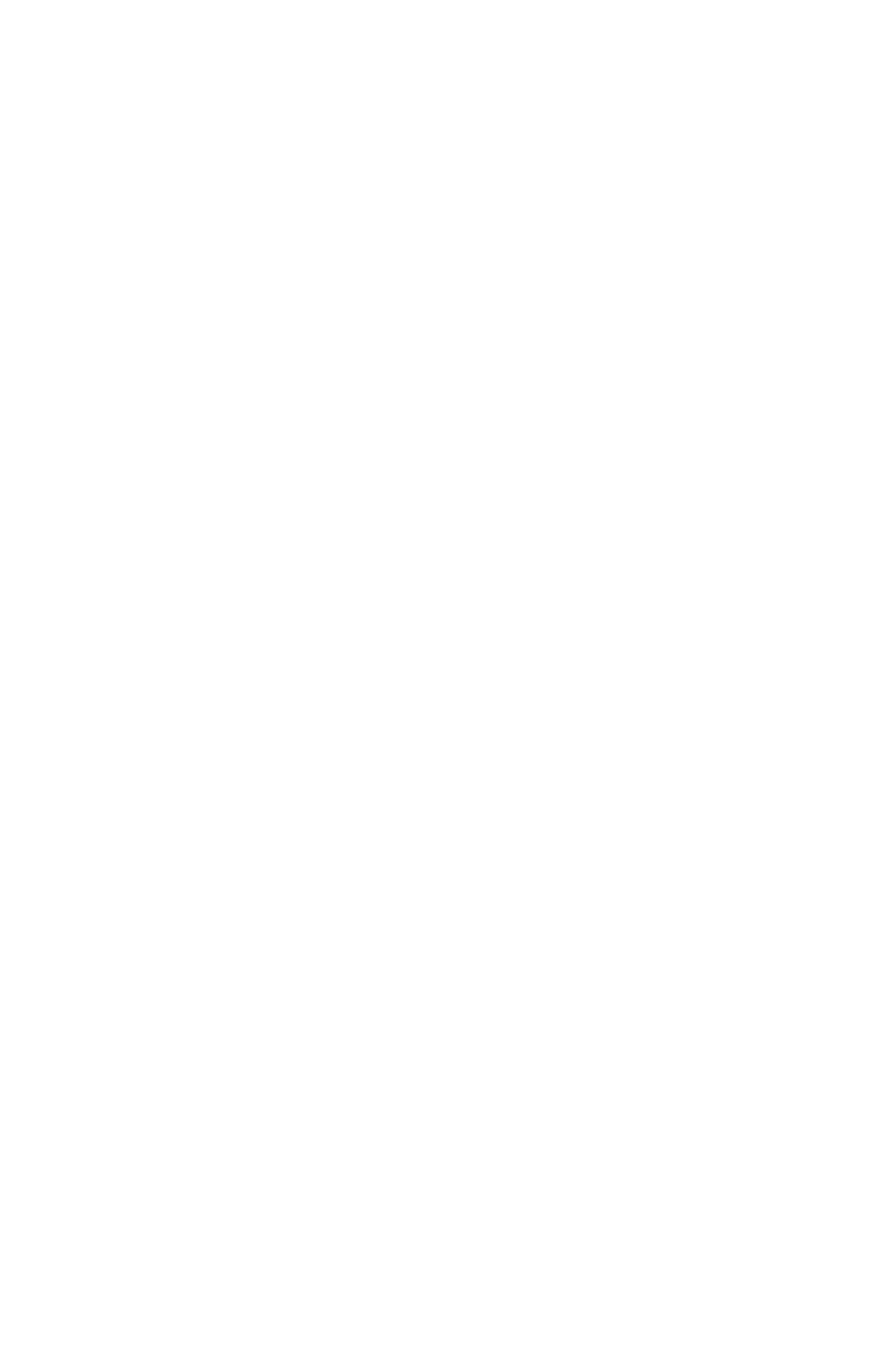
1. Stack Pointer
2. Program Counter

There are a few more parts that do not fit so neatly into categories.

1. Instruction Decoder – The instruction from the **Instruction Register** is decoded here and sent to the ALU and the CU, both of which may require it.
2. Control Bus – The **Clock**, **Write** and **Read** lines connected to the CU make up the Control Bus.
3. Data/Address Bus – The pins **AD0** to **AD7** are multiplexed and can be used both as address lines and as data lines.
4. Address Bus – The pins **A8** to **A15** are used purely as address lines, bringing the total address size up to 16-bits.
5. Controller for Address Latch – This controls the Address Lines.

The only thing remaining in the diagram is the RAM, which is not a part of the processor.

## Pin Diagram



We will not be discussing all the pins, just the important ones.

1. VCC and GND – **Power** comes in at VCC and **noise** goes out at GND (Ground).
2. and – **Read** and **Write** signals go out to memory or I/O devices.
3. /M – A 1 indicates that **memory** is being access; a 0 indicates an **I/O** device is being accessed.
4. INTR and – **Interrupt requests** come in at INTR and **Interrupt Acknowledgements** are granted to processes of higher priority through .
5. HOLD and HLDA – **HOLD signals** come in from processes requesting DMA and **HOLD Acknowledgements** are sent out at HLDA.
6. ALE – **Address Latch Enable** sends 1 to indicate that AD0 – AD7 are being used to send out **addresses**. It sends 0 to indicate that they are being used to send out **data**. This saves space since we do not need to have more pins for a data bus.

## Memory

Memory is divided into two parts, RAM and Registers.

There are a total of **11** registers and 1 **temporary** register. The temporary register is considered separately because it is not mandatory to have. These registers are classified based on their functionality.

1. Accumulator – Holds the latest **result** from the **accumulator**
2. B, C, D, E, H, L – **General purpose** registers; the HL pair can be used for **indirect addressing**
3. Program Counter – The only 16-bit register; holds the address of the **next instruction** that will be executed
4. Instruction Register – Holds the **instruction** currently being executed
5. Stack Pointer – Used during **subroutine calls** and **execution**
6. Address Latch – **Increments/decrements** the address before sending it to the **address buffer**

The RAM on the other hand has memory locations, each with 8 bits of memory. Thus, there is a total of of memory. The addresses go from to in hex.

### Indirect Addressing

We mentioned **indirect addressing** above. To understand this, we first need to understand direct addressing.

MOV B, [FFFA]

ASSEMBLY

Consider the line above. The third-brackets allow us to move the **contents** of the specified memory location to the register B. This is **direct addressing**.

In **indirect addressing**, we store an **actual address** in the HL registers and then copy the contents from that register. The computer sees the address in those registers and then goes to the specified address to get the content.

An address is 16 bits and the registers are 8 bits, so we need to use two registers.

MVI H, FFh  
MIV L, FAh  
MOV B, [HL] *; Indirect Addressing*

ASSEMBLY

### Flag Register

The **Flag Register** has 8 bits, each of which have a different meaning.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| X | X | X | Carry CF | Parity PF | Auxiliary Carry AF | Zero ZF | Sign SF |

Notice that only **five** of the bits are being used, while the left-most three are don’t cares. Each of the five bits is called a **flag**.

1. **Sign Flag** – If the result of the last arithmetic operation has a most-significant byte (MSB) of , it means the number is negative. In this case, this bit has a value of . Otherwise, it has a value of , indicating that the number is positive.
2. **Zero Flag** – If the result of the last operation was , then this bit is set to . Otherwise, it will be .
3. **Auxiliary Carry Flag** – This bit is not accessible by the programmer. It is used by the system during Binary-Coded Decimal (BCD) operations. Essentially, it acts like the carry flag, but only if there is a carry from the 4th bit to the 5th bit. This is because originally, the nibble was used.
4. **Parity Flag** – If the result of the last operation has an even number of s, this bit is set to . Otherwise, it is set to . This flag is used for error checking.
5. **Carry Flag** – If the result of the last operation exceeds 8-bits, then this bit is set to . Otherwise, it is set to .

## Stack Pointer and Stack Memory

For 8085, some portion of the RAM is used as a **Stack**. The Stack is a LIFO structure and can be used by the programmer for **temporary storage** of information.

One odd thing about Stack Memory is that it grows in the **backwards** direction. Basically, it starts at FFFF and goes towards 0000. This keeps it as far from the user program as possible. The exact location where the stack starts can be specified by the programmer.

The **Stack Pointer** points to the location where the top of the stack is located. Imagine that initially, SP FFFF. Now, we move two values into two registers and push those values into the stack.

MVI B, 12h  
MVI C, F3h  
PUSH B *; Pushes the value in register B into the stack*PUSH C

ASSEMBLY

Now, SP FFFD. We can then start popping values. We do this with the POP instruction and by specifying the register into which the value will be popped.

POP B

ASSEMBLY

Now, the register B holds the value F3h. Notice that this was not the value of the register B when we pushed it into the stack. However, the top value of the stack was F3h, and this top value will be the first to get popped. We popped this top value and stored it in the register B. Once this operation is complete, SP FFFE.

If we go on to pop the other value into the register C, we will have swapped the values of B and C with the help of the stack.