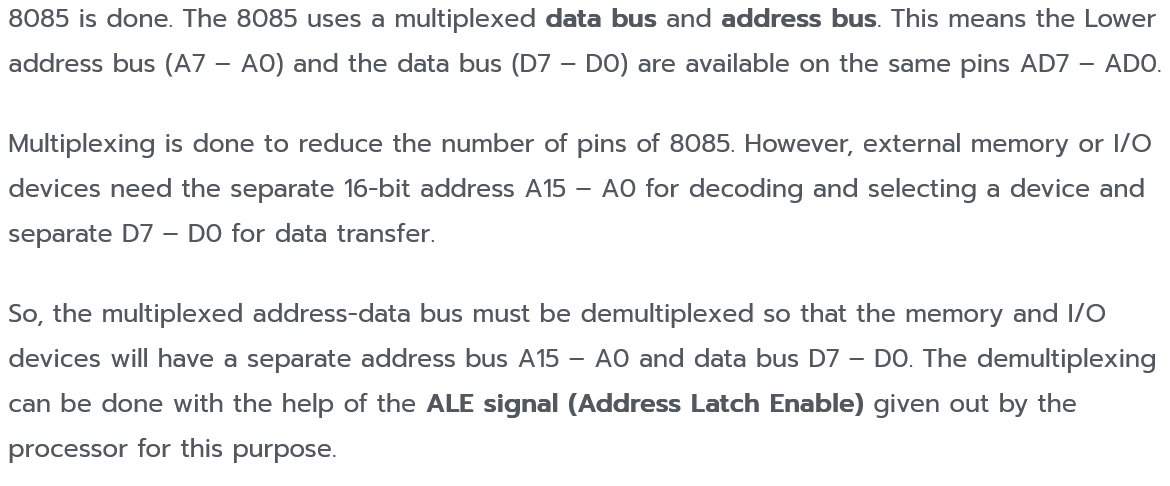
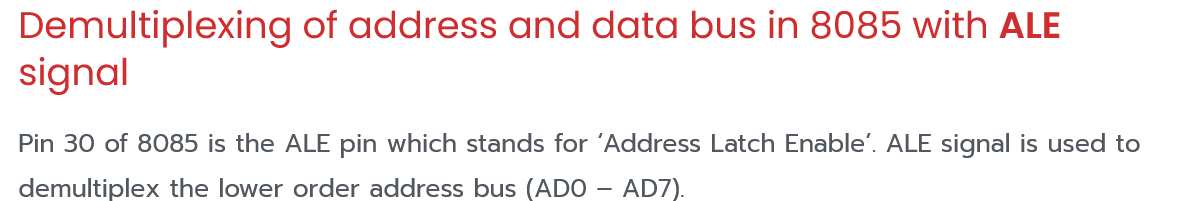
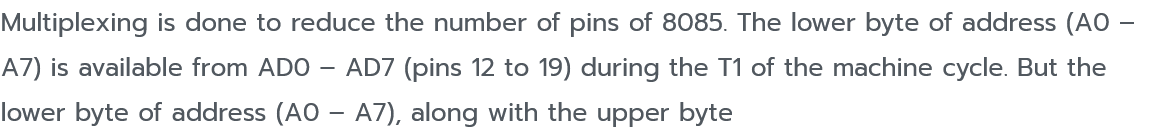
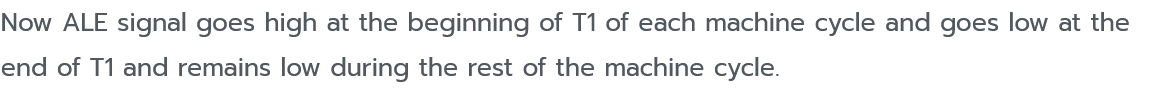
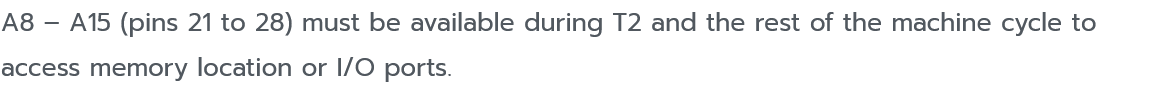
**what is the need of demultiplexing the address/data bus?**

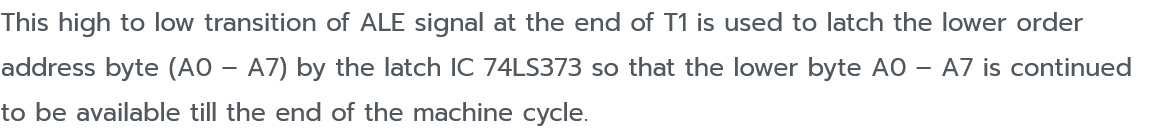


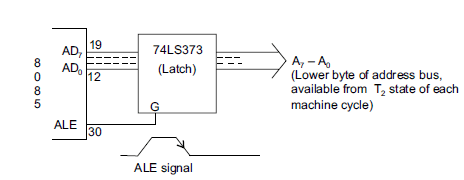


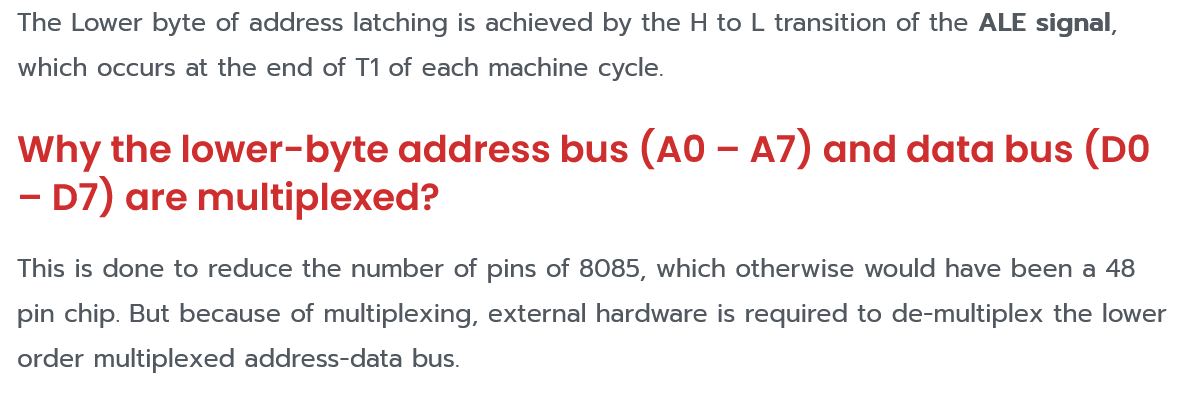










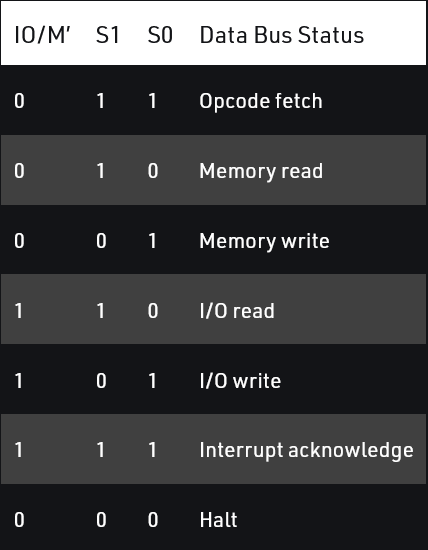


SIGNAL

**READY** − This signal indicates that the device is ready to send or receive data. If READY is low, then the CPU has to wait for READY to go high

This signal is used **to delay the microprocessor Read or Write cycles until a slow-responding peripheral is ready to send or accept data**. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high

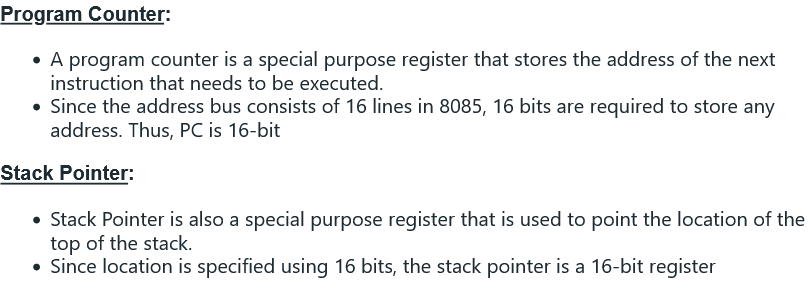
**SO, S1 –** These are status signals. They distinguish the various types of operations such as halt, reading, instruction fetching or writing.



 **HOLD** − This signal indicates that another master is requesting the use of the address and data buses.

 **HLDA (HOLD Acknowledge)** − It indicates that the CPU has received the HOLD request and it will relinquish the bus in the next clock cycle. HLDA is set to low after the HOLD signal is removed.

Why are pc and SP 16 bit register



the 8085 is a 16-bit microprocessor, meaning it uses 16-bit registers, including the program counter (PC) and stack pointer (SP), to store and manipulate data. A 16-bit register allows the 8085 to access a larger amount of memory than an 8-bit register would have,

can the same port address be assigned to two io devices if interfaced in peripheral mapped i/o?

it is possible for two I/O devices to share the same port address if they are interfaced using memory-mapped I/O. In this type of I/O, the processor uses the same memory address space to communicate with both memory and I/O devices. In this case, two I/O devices can share the same port address if they are mapped to different memory addresses.

**How?**

In memory-mapped I/O, the processor uses the same memory address space to communicate with both memory and I/O devices. In this case, two I/O devices can share the same port address if they are mapped to different memory addresses.

For example, if an I/O device is mapped to memory address 0x1000, and another I/O device is mapped to 0x2000, both devices can use the same port address (e.g. 0x00) to communicate with the processor. The processor can differentiate between the two devices by looking at the memory address that the data is being read from or written to.

In this way, memory-mapped I/O allows multiple I/O devices to share the same port address, while still allowing the processor to communicate with each device individually.

**Differentiate between memory mapped I/O and peripheral mapped I/O**

In memory-mapped input/output (I/O), the processor uses the same memory address space to communicate with both memory and I/O devices. This means that I/O devices are accessed using the same memory addresses that are used to access data in memory.

In peripheral-mapped I/O, on the other hand, the processor uses a specific range of memory addresses to communicate with I/O devices. Each I/O device is assigned a unique memory address within this range, and the processor uses this address to access the device.

One key difference between the two types of I/O is that memory-mapped I/O allows multiple I/O devices to share the same port address, while still allowing the processor to communicate with each device individually. In peripheral-mapped I/O, each I/O device must have a unique memory address, so it is not possible for two devices to share the same port address.

Another difference is that memory-mapped I/O allows the processor to directly access I/O devices using the same instructions that are used to access memory, while peripheral-mapped I/O requires the use of special instructions to access I/O devices.

Overall, the choice between using memory-mapped I/O or peripheral-mapped I/O depends on the specific requirements of the system and the design trade-offs that are most important for the application.

**8085 program to convert a hexadecimal number stored in memory location 8200H to it’s corresponding ASCII. Store the result from 8300H**

MVI A, 00H ; initialize accumulator to 0

MOV B, 8200H ; load hexadecimal number from memory location 8200H

ADD B ; add B to A and store the result in A

DAA ; adjust the result in A to form a valid ASCII code

MOV 8300H, A ; store the result in memory location 8300H

HLT ; halt the program

To convert a hexadecimal number to its corresponding ASCII value, we first add 6 to the number. This is because the ASCII codes for the digits 0-9 start at 30H and the codes for the letters A-F start at 41H. By adding 6, we ensure that the result in the accumulator is a valid ASCII code for a hexadecimal digit.

Next, we use the DAA (decimal adjust accumulator) instruction to adjust the result in the accumulator to form a valid ASCII code. This instruction is necessary because the result of the addition may not be a valid ASCII code if the lower nibble (the 4 least significant bits) is greater than 9. For example, if the hexadecimal number in memory location 8200H is B, then adding 6 to it would give us 11 (B(11) + 6 = 17 in decimal which is 11 in hex (17/16 = 1 (result) , 1(quotient))), which is not a valid ASCII code. The DAA instruction adjusts the result in the accumulator to give us the correct ASCII code for B, which is 42H.

After the conversion is complete, the result is stored in memory location 8300H and the program halts.

**CALL and RET are the only instructions of 8085 that uses subroutine. Comment on this statement**

The statement that CALL and RET are the only instructions of 8085 that use subroutines is not correct. In fact, the 8085 microprocessor has several instructions that use subroutines, including JMP, RST, and even some variants of the CALL instruction. These instructions allow the program to transfer control to a subroutine, execute the subroutine, and then return to the main program. Subroutines are useful for organizing and modularizing code, and they can help improve the efficiency and readability of a program.Top of Form

**explain the instruction of 8085 that uses auxiliary carry flag**Bottom of Form

The instruction of the 8085 microprocessor that uses the auxiliary carry flag is the ADC (add with carry) instruction. This instruction adds the contents of a specified memory location or register to the contents of the accumulator, along with the carry flag. The result of the addition is stored in the accumulator.

The auxiliary carry flag (also known as the carry flag) is a bit in the 8085's flag register that is set (or "raised") when a carry is generated from the lower nibble (the four least-significant bits) of the result of the addition. This flag is used to indicate that a carry has occurred and is often used in multi-byte arithmetic operations.

For example, if the accumulator contains the value 0x10 (16 in decimal) and the memory location or register contains the value 0x20 (32 in decimal), the ADC instruction will add these values together, along with the carry flag, to produce a result of 0x30 (48 in decimal). If the carry flag is set, the result of the addition will be 0x31 (49 in decimal). The carry flag will be set in this case because a carry is generated from the lower nibble (the "1" in 0x10 and the "2" in 0x20).

In summary, the ADC instruction of the 8085 microprocessor uses the auxiliary carry flag to indicate whether a carry has occurred during the addition of two values. This flag is used to support multi-byte arithmetic operations and ensure the accuracy of the result.

**Program to clear flag registers of 8085**

MVI A, 0 ; load the value 0 into the accumulator

STC ; set the carry flag

CMC ; complement the carry flag

This program first loads the value 0 into the accumulator, which clears the zero flag. It then sets the carry flag using the STC (set carry) instruction. Finally, it complements the carry flag using the CMC (complement carry) instruction, which clears the carry flag. This will clear all flags in the 8085's flag register, including the auxiliary carry flag, the sign flag, and the parity flag.

Alternatively, you can use the following program to clear the flags:

MVI A, 0 ; load the value 0 into the accumulator

XRA A ; XOR the accumulator with itself

This program uses the XRA (exclusive OR) instruction to XOR the accumulator with itself. Since 0 XOR 0 is 0, this operation will clear the accumulator and all flags in the flag register.

**Interface the 4KB RAM , 2 KB EPROM and 4KB EPROM memory devices in 8085 microprocessor in absolute decoding method**

To interface the 4KB RAM, 2KB EPROM, and 4KB EPROM memory devices in the 8085 microprocessor using the absolute decoding method, you will need to connect the address and data buses of the microprocessor to the memory devices, as well as the control signals (such as RD, WR, and CS) required to access and read from the memory.

First, connect the address bus of the microprocessor to the address input of each memory device. This will allow the microprocessor to send the address of the memory location it wants to access to the memory devices.

Next, connect the data bus of the microprocessor to the data input/output (I/O) of each memory device. This will allow the microprocessor to send data to be written to the memory or receive data from the memory.

Finally, connect the control signals of the microprocessor (such as RD, WR, and CS) to the corresponding inputs of each memory device. These signals are used to control the access and read/write operations of the memory.

To use the absolute decoding method, you will need to assign a specific range of addresses to each memory device. For example, you can assign the range 0x0000-0x0FFF to the 4KB RAM, 0x1000-0x17FF to the 2KB EPROM, and 0x1800-0x1FFF to the 4KB EPROM. When the microprocessor wants to access a specific memory location, it will send the address of that location on the address bus. The memory devices will decode the address and only the device with the matching address range will respond to the access request.

For example, if the microprocessor sends the address 0x1200 on the address bus, only the 2KB EPROM will respond because its address range is 0x1000-0x17FF and 0x1200 falls within that range. The 4KB RAM and 4KB EPROM will not respond because their address ranges do not match the address sent by the microprocessor.

In this way, the absolute decoding method allows you to map the memory devices to specific ranges of addresses and control access to each device independently. This can be useful for organizing and managing the memory resources of the microprocessor.

**Explain the role of stack in executing subroutine**

When a subroutine is called, its instructions are pushed onto the top of the stack. The subroutine then executes, and when it is finished, its instructions are popped off the stack and control is returned to the point where the subroutine was called.

This use of a stack allows for the efficient execution of subroutines, because it provides a way to keep track of the order in which the subroutines were called, and ensures that they are executed in the correct order. It also allows for the subroutines to be nested (i.e. for one subroutine to call another subroutine), because the stack can keep track of the execution of each subroutine independently.