1.1 Introduction to Microprocessors and Interfacing

There is no doubt that the microcomputer revolution will continue into the future and many will be required to specify and integrate microprocessors into products or systems in their own disciplines. Therefore, well-designed flexible interfaces will be required to ensure compatibility with other equipments and to extend design options.

Interfaces are the last items to be seriously considered in the race of new technology, and it deals with the systematic study of microprocessor interfaces and their applications in many diversified fields.

With the advent of the first 4-bit microprocessor 4004 from Intel Corporation in 1971, there has been a silent revolution in the domain of digital system design, which has shaken many facets of the current technological progress. In the last 28 years the world has seen an evolution of microprocessors, whose impact on today's technological scenario is phenomenal.

This evolution was possible because of the tremendous advances in the semiconductor process technology. The first microprocessor 4004 contained only ten thousand transistors while the component density increased more than threefold in less than a decade's time. Immediately after the introduction of the 4004, Intel introduced the first eight bit microprocessor 8008 in 1972; these processors were, however, not successful because of their inherent limitations. In 1974, Intel released the first general-purpose 8-bit microprocessor 8080. This CPU also was not functionally complete and the first 8-bit functionally complete CPU 8085 was introduced in 1977.
The major limitations of these 8-bit microprocessors are their limited memory addressing capacity, slow speed of execution, limited number of scratchpad registers and non-availability of complex instruction set and addressing modes. Another important point to be mentioned here is that 8085 does not support adequate pipelining or parallelism, which is so important for enhancing the speed of computation. For example, the non-availability of any instruction queue in an 8085 CPU leads to a situation where the fetching of opcode and operands along with the execution takes place in an absolutely sequential manner. The first 16-bit CPU from Intel was a result of the designers' efforts to produce a more powerful and efficient computing machine. The designers of 8086 CPU had taken note of the major limitations of the previous generations of the 8-bit CPUs. The 8086 contain a set of 16-bit general-purpose registers, support a 16-bit ALU, a rich instruction set and provide segmented memory addressing scheme. The introduction of a set of segment registers for addressing the segmented memory in 8086 was indeed a major step in the process of evolution. All these features made this 16-bit processor a more efficient CPU.

The development of IBM PC started in July 1980, and precisely after one year, the first machine based on Intel 8088 CPU (which is functionally equivalent to 8086 but supports only 8-bit external data bus) with 1 or 2 floppy disk drives, a keyboard and a monochrome monitor was announced in August 1981. The machine operating system was an early version of operating system MS-DOS from Microsoft. In March 1983, a new version of IBM PC called PC-XT was introduced with ten-megabyte hard disk, one double side double density floppy disk drive, keyboard, monitor and asynchronous communication adapter. In fact, the introduction of IBM PCs in 1980s had, to a large extent, produced a profound impact on the evolution of microprocessors. With the introduction of each new generation of micro-processors, the performance of the Personal Computers has also been enriched. The major limitation in 8086 was that it did not have the
memory management and protection capabilities, which were considered an extremely important feature, deemed to be an integral part of a CPU of the eighties. 80286 was the first CPU to possess the ability of memory management, privilege and protection. However, the 80286 CPU also had a limitation on the maximum segment size supported by it (only 64 Kb). Another limitation of 80286 was that, once it was switched into protected mode, it was difficult to get it back to real mode. The only way of reverting it to the real mode was to reset the system.

In the mid eighties the more computationally demanding problems necessitated the development of still faster CPUs. Thus appeared 80386, which was the first 32-bit CPU from Intel. The memory management capability of 80286 was enhanced to support virtual memory, paging and four levels of protection. The design of 80386 circumvented this problem. Moreover, the maximum segment size in 80386 was enhanced and this could be as large as 4 Gb with 80386 supporting as many as 16384 segments. The 80386 along with its math coprocessor 80387, provided a high speed environment. 80486 was designed with an integrated math coprocessor. After getting integrated, the speed of execution of mathematical operations enhanced three folds. Also for the first time an 8 Kb four-way set associative code and data cache was introduced in 80486. A five-stage instruction pipelining was also introduced.

The earlier generation CPUs supported rather crude instruction sets. It was not expected that the programmers those days would write large machine code programs. A single high-level instruction might be compiled into ten or even hundred machine code operations. In the course of evolution from the early 8-bit CPUs, the trend was to design CPUs, which could support more and more complex instructions at the assembly language level. Designers of complex instruction set computers (CISC) wanted to reduce this gap.

Since the early days of microprocessor development the designers have tried to make them more powerful by designing more complex instructions. But
then some of these powerful instructions and addressing modes were hardly used by the programmers. In fact some of these instructions' logic took up a large part of the microprocessors' silicon chip. The reduced instruction set computer (RISC) designers observed that the data movement type of machine instructions are frequently executed by the CPU. They have optimized the CPUs to execute these instructions rapidly. RISC provided a regular set of instructions having the same format with a lot of pipelining.

To improve the processor's performance, the possible ways are suggested below.

(a) Increasing the processor and system clock rate.
(b) Optimizing and improving the instruction set.
(c) Executing multiple instructions in one cycle and incorporating parallelism in the CPU architecture.

The first option is applicable both to CISC and RISC processors. The second option is primarily for CISC but is applicable to RISC as well. The third option is more suited to RISC CPUs. Ever since the appearance of commercially available RISC CPUs, there has been a debate over the performance of RISC versus CISC. The RISC architects argue that their instructions may be executed in a single cycle and thus take less time than is taken by a CISC CPU. This is because of pipelining, reduction of instructions to a simple operation and synthesis of complex operations with compiler generated code sequences. When RISC machines first arrived in the market, CISC processors were performing at 6-10 cycles per instruction, while the RISC CPUs could execute a set of simpler instructions in one cycle and offer better performance. Many of the CISC processors have subsequently used many features of RISC.

8086 microprocessor has a much more powerful instruction set along with the architectural developments which imparted substantial programming flexibility and improvement in speed over the 8-bit microprocessors.
The peripheral chips designed earlier for 8085 were compatible with microprocessor 8086 with slight or no modifications. Though there is a considerable difference between the memory addressing techniques of 8085 and 8086, the memory interfacing technique is similar. But includes the use of a few additional signals.

1.2 **8086 Internal Architecture**

As shown by the block diagram in the Figure, the 8086 CPU is divided into two independent functional parts, the bus Interface unit or BIU, and the execution unit or EU. Dividing the work between these two units speeds up processing.

![8086 Internal Block Diagram](image)

The bus interface unit contains the circuit for physical address calculations and a pre-decoding instruction byte queue (6 bytes long). The bus interface unit makes the system bus signals available for external interfacing of the devices. In other words, this unit is responsible for establishing communications with...
external devices and peripherals including memory via the bus. As already stated, the 8086 addresses a segmented memory. The complete physical address, which is 20-bits long, is generated using segment and offset registers, each 16-bits long.

For generating a physical address from contents of these two registers, the content of a segment register also called as segment address is shifted left bitwise four times and to this result, content of an offset register also called as offset address is added, to produce a 20-bit physical address. For example, if the segment address is $1005H$ and the offset is $5555H$, then the physical address is calculated as below.

\[
\begin{align*}
\text{Segment address} & \quad 1005H \\
\text{Offset address} & \quad 5555H \\
\text{Segment address} & \quad 1005H \quad \text{--} \quad 0001 \ 0000 \ 0000 \ 0101 \\
\text{Shifted by 4 bit positions} & \quad 0001 \ 0000 \ 0000 \ 0101 \ 0000 \ + \\
\text{Offset address} & \quad 0101 \ 0101 \ 0101 \ 0101 \\
\text{Physical address} & \quad 001 \ 0101 \ 0101 \ 1010 \ 0101 \\
\end{align*}
\]

Thus the segment addressed by the segment value $1005H$ can have offset values from $0000H$ to $FFFFH$ within it, i.e. maximum 64K locations may be accommodated in the segment. Thus the segment register indicates the base address of a particular segment, while the offset indicates the distance of the required memory location in the segment from the base address. Since the offset is a 16-bit number, each segment can have a maximum of 64K locations. The bus interface unit has a separate adder to perform this procedure for obtaining a physical address while addressing memory. The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP or an immediate 16-bit value, depending upon the addressing mode.
1.2.1 THE EXECUTION UNIT
Control Circuitry, Instruction Decoder, And ALU

As shown in the Figure, the EU contains control circuitry, which directs internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions, which the EU carries out. The EU has a 16-bit arithmetic logic unit which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

1.2.1.1. Flag Register

A flag is a flip-flop, which indicates some condition produced by the execution of an instruction, or controls certain operations of the EU. A 16-bit flag register in the EU contains nine active flags. Following figure shows the location of the nine flags in the flag register.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>D</td>
<td>I</td>
<td>T</td>
<td>S</td>
<td>Z</td>
<td>X</td>
<td>Ac</td>
<td>X</td>
<td>P</td>
<td>X</td>
<td>Cy</td>
</tr>
</tbody>
</table>

O — Overflow flag
D — Direction flag
I — Interrupt flag
T — Trap flag
S — Sign flag
Z — Zero flag
Ac — Auxiliary carry flag
P — Parity flag
Cy — Carry flag
X — Not used

Six of the nine flags are used to indicate some condition produced by an instruction. For example, a flip-flop called the carry flag will be set to a 1 if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a 0. The EU thus effectively runs up a "flag" to tell you that a carry was produced.

The six conditional flags in this group are the carry flag (CF), the parity flag (PF), the auxiliary carry flag (AF), the zero flag (ZF), the sign flag (SF), and the overflow flag (OF). Certain 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instruction.
S-Sign Flag: This flag is set, when the result of any computation is negative. For signed computations, the sign flag equals the MSB of the result.

Z-Zero Flag: This flag is set, if the result of the computation or comparison performed by the previous instruction/instructions is zero.

P-Parity Flag: This flag is set to 1, if the lower byte of the result contains even number of 1s.

C-Carry Flag: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

T-Trap Flag: If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

I-interrupt Flag: If this flag is set, the maskable interrupts are recognized by the CPU, otherwise, they are ignored.

D-Direction Flag: This is used by string manipulation instructions. If this flag bit is '0' the string is processed beginning from the lowest address to the highest address, i.e. auto-incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e. auto-decrementing mode. We will describe string manipulations later in chapter 2 in more details.

AC-Auxiliary Carry Flag: This is set, if there is a carry from the lowest nibble, i.e. bit three, during addition or borrow for the lowest nibble, i.e. bit three, during subtraction.

O-Overflow Flag: This flag is set, if an overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register. For example, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e. the result is of more than 7-bits in size in case of 8-bit signed operations and more than 15-bits in size in case of 16-bit signed operations, and then the overflow flag will be set.
The three remaining flags in the flag register are used to control certain operations of the processor. These flags are different from the six conditional flags described above in the way they are set or reset. The six conditional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation. The control flags are deliberately set or reset with special instructions you put in your program. The three control flags are the trap flag (TF), which is used for single stepping through a program; the interrupt flag (IF), which is used to allow or prohibit the interruption of a program; and the direction flag (DF), which is used with string instructions.

1.2.1.2. General-Purpose Registers

The EU has eight general-purpose registers, labeled AH, AL, BH, BL, CH, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the Accumulator. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. The AH-AL pair is referred to as the AX register, the BH-BL pair is referred to as the BX register, the CH-CL pair is referred to as the CX register, and the DH-DL pair is referred to as the DX register.

The 8086 general-purpose register set is very similar to those of the earlier-generation 8080 and 8085 microprocessors. It was designed this way so that the many programs written for the 8080 and 8085 could easily be translated to run on the 8086 or the 8088. The advantage of using internal registers for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in external memory. Now let's look at the features of the BIU.
1.2.2 The Bus Interface Unit

1.2.2.1. THE INSTRUCTION QUEUE

While the EU is decoding an instruction or executing an instruction, which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions. The BIU stores these prefetched bytes in a first-in-first-out register set called a queue. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. Except in the cases of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called **pipelining**.

1.2.2.2. SEGMENT REGISTERS

The 8086 BIU sends out 20-bit addresses, so it can address any of $2^{20}$ or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536-byte (64-Kbyte) segments within this 1,048,576-byte (1-Mbyte) range. Four segment registers in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time.

The four segment registers are the code segment (CS) register, the stack segment (SS) register, the extra segment (ES) register, and the data segment (DS) register. Figure above shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs, which do not need all 64 Kbytes in each segment they can overlap.

To repeat, then, a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from
which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest 4 bits (nibble) of the 20-bit starting address for a segment.

![Diagram showing memory addressing and segment placement](image)

*Fig: One way four 64-Kbyte segments might positioned within 1-Mbyte address space of an 8086.*

If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits. This constraint was put on the location of segments so that it is only necessary to store and manipulate 16-bit numbers when working with the starting address of a segment. The part of a segment starting address stored in a segment register is often called the segment base.

A stack is a section of memory set aside to store addresses and data while a subprogram executes. The stack segment register is used to hold the upper 16
bits of the starting address for the program stack. The extra segment register and the data segment are used to hold the upper 16 bits of the starting addresses of two memory segments that are used for data.

![Diagram](image)

**Fig:** Addition of IP to CS to produce the physical address of the code byte. (a) Diagram and (b) Computation.

*1.2.2.3. INSTRUCTION POINTER*

The next feature to look at in the BIU is the Instruction pointer (IP) register. As discussed previously, the code segment registers hold the upper 16 bits of the starting address of the segment from which the BIU is currently fetching instruction code bytes. The instruction pointer register holds the 16-bit address, or offset, of the next code byte within this code segment. The value contained in the IP is referred to as an offset because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address sent out by the BIU. Figure above shows in diagram form how this works. The CS register points to the base or start of the current code segment. The IP contains the distance or offset from this base address to the next instruction byte to be fetched. Above figure shows how the 16-bit offset in IP is added to the 16-bit segment base address in CS to produce the 20-bit physical address. Notice that the two 16-bit numbers are not added directly in line,
because the CS register contains only the upper 16 bits of the base address for the
code segment. The BIU automatically inserts zeros for the lowest 4 bits of the
segment base address.

If the CS register, for example, contains 348AH, you know that the starting
address for the code segment is 348A0H. When the BIU adds the offset of 4214H
in the IP to this segment base address, the result is a 20-bit physical address of
38AB4H. An alternative way of representing a 20-bit physical address is the
segment base : offset form. For the address of a code byte, the format for this
alternative form will be CS:IP. As an example of this, the address 38AB4H, can
also be represented as 348A:4214.

To summarize, then, the CS register contains the upper 16 bits of the
starting address of the code segment in the 1-Mbyte address range of the 8086.
The instruction pointer register contains a 16-bit offset, which tells where in that
64-Kbyte code segment the next instruction byte is to be fetched from. The actual
physical address sent to memory is produced by adding the offset contained in
the IP register to the segment base represented by the upper 16 bits in the CS
register.

Any time the 8086 accesses memory, the BIU produces the required 20-bit
physical address by adding an offset to a segment base value represented by the
contents of one of the segment registers.

STACK SEGMENT REGISTER AND STACK POINTER REGISTER

A stack, remember, is a section of memory set aside to store addresses and
data while a subprogram is executing. The 8086 allow you to set aside an entire
64-Kbyte segment as a stack. The upper 16 bits of the starting address for this
segment are kept in the stack segment register. The stack pointer (SP) register in
the execution unit holds the 16-bit offset from the start of the segment to the
memory location where a word was most recently stored on the stack. The
memory location where a word was most recently stored is called the top of
stack. Figure above, shows this in diagram form.
1.2.3. POINTER AND INDEX REGISTERS IN THE EXECUTION UNIT

In addition to the stack pointer register (SP), the EU contains a 16-bit base pointer (BP) register. It also contains a 16-bit source index (SI) register and a 16-bit destination Index (DI) register. These three registers can be used for temporary storage of data just as the general-purpose registers described above. However, their main use is to hold the 16-bit offset of a data word in one of the segments. SI, for example, can be used to hold the offset of a data word in the data segment. The physical address of the data in memory will be generated in this case by adding the contents of SI to the segment base address represented by the 16-bit number in the DS register.

Fig: Addition of data segment (DS) register and effective address (BX) to produce the physical address of the data byte.

1.3 SIGNAL DESCRIPTIONS OF 8086

The microprocessor 8086 is a 16-bit CPU available in three clock rates, i.e. 5, 8 and 10 MHz, packaged in a 40 pin CERDIP or plastic package. The 8086
operates in single processor or multiprocessor configurations to achieve high performance. The pin configuration is shown in the following figure. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode) configuration.

The 8086 signals can be categorized in three groups. The first are the signals having common functions in minimum as well as maximum mode, the second are the signals, which have special functions for minimum mode, and the third are the signals having special functions for maximum mode.

The following signal descriptions are common for both the minimum and maximum modes.

**AD15-AD0** These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T, state, while the data is available on the data bus during T2, T3, T4, and Tw. Here T1, T2, T3, T4 and Tw are the clock states of a machine cycle. Tw is a wait state. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

![The Pin Configuration of 8086](image)

**A19/S6, A18/S5, A17/S4, A16/S3** These are the time multiplexed address and status lines. During T1, these are the most significant address lines for memory
operations. During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for $T_2$, $T_3$, $T_w$, and $T_4$. The status of the interrupt enable flag bit (displayed on $S_5$) is updated at the beginning of each clock cycle.

<table>
<thead>
<tr>
<th>$S_4$</th>
<th>$S_3$</th>
<th>Indications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Alternate Data</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Stack</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Code or None</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Data</td>
</tr>
</tbody>
</table>

Table: Bus High Enable/Status

The $S_4$ and $S_3$ combinedly indicate which segment register is presently being used for memory accesses as shown in Table above. These lines float to tri-state off (tristated) during the local bus hold acknowledge. The status line $S_6$ is always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

**BHE/S7-Bus High Enable/Status** The bus high enable signal is used to indicate the transfer of data over the higher order ($D_{15}$-$D_8$) data bus as shown in Table below. It goes low for the data transfers over $D_{15}$-$D_8$ and is used to derive chip selects of odd address memory bank or peripherals. $BHE$ is low during $T_1$ for read, write and interrupt acknowledge cycles, whenever a byte is to be transferred on the higher byte of the data bus. The status information is available during $T_2$, $T_3$ and $T_4$. The signal is active low and is tristated during 'hold'. It is low during $T_1$ for the first pulse of the interrupt acknowledge cycle.

<table>
<thead>
<tr>
<th>BHE</th>
<th>$A_0$</th>
<th>Indications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Whole Word</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Upper Byte from or to Odd Address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Lower Byte from or to Even Address</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
</tbody>
</table>

**RD-Read** Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. $RD$ is active low and shows the
state for T₂, T₃, Tₜ of any read cycle. The signal remains tristated during the 'hold acknowledge'.

**READY** This is the acknowledgement from the slow devices or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The signal is active high.

**INTR**-interrupt Request This is a level triggered input. This is sampled during the last clock cycle of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resetting the interrupt enable flag. This signal is active high and internally synchronized.

**TEST** This input is examined by a 'WAIT' instruction. If the TEST input goes low, execution will continue, else, the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

**NMI** Non-maskable Interrupt This is an edge-triggered input which causes a Type2 interrupt. The NMI is not maskable internally by software. A transition from low to high initiates the interrupt response at the end of the current instruction. This input is internally synchronized.

**RESET** This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronized.

**CLK** Clock Input The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle. The range of frequency for different 8086 versions is from 5MHz to 10MHZ.

**Vcc** +5V power supply for the operation of the internal circuit. GND ground for the internal circuit.
MN/MX The logic level at this pin decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode. The following pin functions are for the minimum mode operation of 8086.

M/I/O - Memory/IO This is a status line logically equivalent to 92 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous T4 and remains active till final T4 of the current cycle. It is tristated during local bus "hold acknowledge".

INTA - Interrupt Acknowledge This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during T2, T3 and Tw of each interrupt acknowledge cycle.

ALE-Address Latch Enable This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.

DT/R Data Transmit/Receive
This output is used to decide the direction of data flow through the transreceivers (bi-directional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to S, in maximum mode. Its timing is the same as M/I/O. This is tristated during 'hold acknowledge'.

DEN-Data Enable This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bi-directional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T2 until the middle of T4 DEN is tristated during 'hold acknowledge' cycle.

HOLD, HLDA - Hold/Hold Acknowledge When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal
on HLDA pin, in the middle of the next clock cycle after completing the current bus (instruction) cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and it should be externally synchronized. If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T4 provided:

1. The request occurs on or before T2 state of the current cycle.
2. The current cycle is not operating over the lower byte of a word (or operating on an odd address).
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A Lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode.

The following **pin functions** are applicable for **maximum mode** operation of 8086.

**$S_2, S_1, S_0$ - Status Lines** These are the status lines which reflect the type of operation, being carried out by the processor. These become active during T4 of the previous cycle and remain active during T, and T2 of the current bus cycle. The status lines return to passive state during T3 of the current bus cycle so that they may again become active for the next bus cycle during T4. Any change in these lines during T3 indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in the Table.

<table>
<thead>
<tr>
<th>$S_2$</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>Indications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Interrupt Acknowledge</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Read I/O Port</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Write I/O Port</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Halt</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Code Access</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read Memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Write Memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Passive</td>
</tr>
</tbody>
</table>

**LOCK** This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the $\overline{LOCK}$ signal is low. The $\overline{LOCK}$ signal is
activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during 'hold acknowledge'. When the CPU is executing a critical instruction, which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

**QS<sub>1</sub>, QS<sub>0</sub>-Queue Status** These lines give information about the status of the code-prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table below.

<table>
<thead>
<tr>
<th>QS&lt;sub&gt;1&lt;/sub&gt;</th>
<th>QS&lt;sub&gt;0&lt;/sub&gt;</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No operation</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>First byte of opcode from the queue</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Empty queue</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Subsequent byte from the queue</td>
</tr>
</tbody>
</table>

This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of pipelined processing of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus even the largest (6-bytes) instruction can be prefetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as instruction pipelining.

At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even. The first byte is a complete opcode in case of some instructions (one byte opcode instruction) and it is a part of opcode, in case of other
instructions (two byte long opcode instructions), the remaining part of opcode may lie in the second byte. But invariably the first byte of an instruction is an opcode. These opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied.

![Diagram of Instruction QUEUE Operation](image)

Fig: The Instruction QUEUE Operation

The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is single opcode byte, the next bytes are treated as data bytes depending upon the decoded instruction length, otherwise the next byte in the queue is treated as the second byte of the instruction opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two
bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.

The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture, there are two separate units, namely, execution unit and bus interface unit, while the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

**RQ/GT₀, RQ/GT₁-Request/Grant**

Other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor’s current bus cycle, use these pins. Each of the pins is bi-directional with RQ/GT₀ having higher priority than RQ/GT₁. RQ/GT pins have internal pull-up resistors and may be left unconnected. The request/grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.
2. During T₄ (current) or T₅ (next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at next clock cycle. The CPU’s bus interface unit is likely to be disconnected from the local bus of the system.
3. A one clock wide pulse from another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle.

Thus each master-to-master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The
request and grant pulses are active low. For the bus requests those are received while 8086 is performing memory or 1/0 cycle, the granting of the bus is governed by the rules as discussed in case of HOLD, and HLDA in minimum mode.

1.4 PHYSICAL MEMORY ORGANISATION

In an 8086 based system, the 1-Mbytes memory is physically organized as odd bank and even bank, each of 512Kbytes, addressed in parallel by the processor. Byte data with even address is transferred on D7 – D0, while the byte data with odd address is transferred on D15- D8 bus lines. The processor provides two enable signals, BHE and A0 for selection of either even or odd or both the banks. The instruction stream is fetched from memory as words and is addressed internally by the processor as necessary. In other words, if the processor fetches a word (consecutive two bytes) from memory, there are different possibilities, like:

1. Both the bytes may be data operands.
2. Both the bytes may contain opcode bits.
3. One of the bytes may be opcode while the other may be data.

Fig: 8086 Memory Banks
All the above possibilities are taken care of by the internal decoder circuit of the microprocessor. The opcodes and operands are identified by the internal decoder circuit, which further derives the signals those act as input to the timing and control unit. The timing and control unit then derives all the signals required for execution of the instruction.

In referring word data, the BIU requires one or two memory cycles, depending upon whether the starting byte is located at an even or odd address. It is always better to locate the word data at an even address. To read or write a complete word from/to memory, if it is located at an even address, only one read or write cycle is required. If the word is located at an odd address, the first read or write cycle is required for accessing the lower byte while the second one is required for accessing the upper byte. Thus two bus cycles are required, if a word is located at an odd address. It should be kept in mind that while initializing the structures like stack they should be initialized at an even address for efficient operation.

Certain locations in memory are reserved for specific CPU operations. The locations from FFFF0H to FFFFFH are reserved for operations including jump to initialization, programme and I/O-processor initialization. The locations 00000H to 003FFH are reserved for interrupt vector table. The interrupt structure provides space for a total of 256 interrupt vectors. The vectors, i.e. CS and IP for each interrupt routine requires 4 bytes for storing it in the interrupt vector table. Hence 256 types of interrupt require $256 \times 4 = 03FFH$ (1Kbyte) locations for the complete interrupt vector table.

**1.5. GENERAL BUS OPERATION**

The 8086 has a combined address and data bus commonly referred to as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40-pin standard DIP package. The bus can be demultiplexed using a few latches and transreceivers, whenever required. In the following text, we will discuss a general bus operation cycle.
Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as \( T_1 \), \( T_2 \), \( T_3 \) and \( T_4 \). The address is transmitted by the processor during \( T_1 \). It is present on the bus only for one cycle. During \( T_2 \), i.e. the next cycle, the bus is tristated for changing the direction of bus for the following data read cycle. The data transfer takes place during \( T_3 \) and \( T_4 \). In case, an addressed device is slow and shows 'NOT READY' status the wait states \( T_w \) are inserted between \( T_3 \) and \( T_4 \). These clock states during wait period are called idle states (\( T_i \)), wait states (\( T_w \)) or inactive states. The processor uses these cycles for internal housekeeping. The address latch enable (ALE) signal is emitted during \( T \), by the processor (minimum mode) or the bus controller (maximum mode) depending upon the status of the MN/MX input. The negative edge of this ALE pulse is used to separate the address and the data or status information.

In maximum mode, the status lines \( S_0 \), \( S_1 \) and \( S_2 \) are used to indicate the type of operation as discussed in the signal description section of this chapter. Status bits \( S_3 \) to \( S_7 \) are multiplexed with higher order address bits and the BHE signal. Address is valid during \( T \), while the status bits \( S_3 \) to \( S_7 \) are valid during \( T_2 \) through \( T_4 \). The figure below shows a general bus operation cycle of 8086.
1.6 I/O ADDRESSING CAPABILITY

The 8086/8088 processors can address up to 64K I/O byte registers or 32K word registers. The limitation is that the address of an I/O device must not be greater than 16 bits in size; this means that a maximum number of $2^{16}$, i.e. 64Kbyte I/O devices may be accessed by the CPU. The I/O address appears on the address lines $A_0$ to $A_3$ for one clock cycle ($T_1$). It may then be latched using the ALE signal. The upper address lines ($A_{16} - A_{19}$) are at logic 0 level during the I/O operations.

The 16-bit register DX is used as 16-bit I/O address pointer, with full capability to address up to 64K devices. In this case, the I/O ports are addressed in the same manner as memory locations in the based addressing mode using BX. In memory mapped I/O interfacing, the I/O device addresses are treated as memory locations in page 0, i.e. segment address 0000H. Even addressed bytes are transferred on $D_7 - D_0$ and odd addressed bytes are transferred on $D_8 - D_{15}$ lines. While designing any 8 bit I/O system around 8086, care must be taken that all the byte registers in the system should be addressed even.

SPECIAL PROCESSOR ACTIVITIES

Processor Reset and Initialization

When logic I is applied to the RESET pin of the microprocessor, it is reset and it remains in the reset state till logic 0 is again applied to the RESET pin. The 8086 terminate the ongoing operation on the positive edge of the reset signal. When the negative edge is detected, the reset sequence starts and is continued for nearly 10 clock cycles. During this period, all the internal register contents are set to 0000H except CS is set to value F000H and IP to value FFF0H. Thus the execution again starts from the physical address FFFF0H. Due to this, the EPROM in an 8086 system is interfaced so as to have the physical memory locations FFFF0H to FFFFFH in it, i.e. at the end of the map.

For the reset signal to be accepted by 8086, it must be high for at least 4 clock cycles. From the instant the power is on, the reset pulse should not be
applied to 8086 before 50,us to allow proper initialization of 8086. In the reset state, all the 3-state outputs are tristated. Status signals are active in idle state for the first clock cycle after the reset becomes active, and then floats to tristate. The ALE and HLDA lines are driven low during the reset operation.

Non-maskable interrupt enable request, which appears before the second clock after the end of the reset operation, will not be served. For the NMI request to be served, it must appear after the second clock cycle during reset initialisation or later. If a HOLD request appears immediately after RESET, it will be immediately served after initialisation, before execution of any instruction.

HALT

When the processor executes a HLT instruction, it enters the 'halt' state. Before entering 'halt' state, it indicates that it is entering 'halt' state in two ways, depending upon whether it is in minimum or maximum mode. When the processor is in minimum mode and wants to enter halt state, it issues an ALE pulse but does not issue any control signal. When the processor is in maximum mode and wants to enter halt state, it puts the HALT status on \( S_2, S_1 \) and \( S_0 \) pins and then the bus controller issues one ALE pulse but no qualifying signal, i.e. no appropriate address or control signals are issued onto the bus. Only an interrupt request or reset will force the 8086 to come out of the 'halt' state. Even the HOLD request cannot force the 8086 out of 'halt' state.

TEST and Synchronization with External Signals

Besides the interrupt, hold and general I/O capabilities, the 8086 has an extra facility of the TEST signal. When the CPU executes a WAIT instruction, the processor preserves the contents of the registers, before execution of the WAIT instruction, and the CPU waits for the TEST input pin to go low. If the TEST pin goes low, it continues further execution; otherwise, it keeps on waiting for the TEST pin to go low. For the TEST signal to be accepted, it must be low for at least 5 clock cycles. The activity of waiting does not consume any bus cycle. The processor remains in idle state while waiting. While waiting, any 'HOLD' request
from an external device may be served. If an interrupt occurs when the processor is waiting, it fetches the wait instruction once more, executes it, and then serves the interrupt. After returning from the interrupt, it fetches the wait instruction once more and continues with the 'wait' state.

Thus the execution of the portion of a program, which appears in the program after WAIT instruction can be synchronized with an external signal connected with the TEST input.

1.7. THE PROCESSOR 8088

The launching of the processor 8086 is seen as a remarkable step in the development of high speed computing machines. Before the introduction of 8086, most of the circuits required for the different applications in computing and industrial control fields were already designed around the 8-bit processor 8085. The 8086 imparted tremendous flexibility in the programming as compared to 8085. So naturally, after the introduction of 8086, there was a search for a microprocessor chip which has the programming flexibility like 8086 and the external interface like 8085, so that all the existing circuits built around 8085 can work as before, with this new chip. The chip 8088 was a result of this demand. The microprocessor 8088 has all the programming facilities that 8086 has, along with some hardware features of 8086, like 1-Mbyte memory addressing capability, operating modes (MN/MX), interrupt structure etc. However 8088, unlike 8086, has 8-bit data bus. This feature of 8088 makes the circuits, designed around 8085, compatible with 8088, with little or no modification.

All the peripheral interfacing schemes with 8088 are the same as those for the 8-bit processors. The memory and I/O addressing schemes are now exactly similar to 8085 schemes except for the increased memory (1-Mbyte) and I/O (64Kbyte) capabilities. The architecture shows the developments in 8088 over 8086.

**Architecture of 8088:** The register set of 8088 is exactly the same as in to 8086. The architecture of 8088 is also similar to 8086 except for two changes;
a) 8088 has 4-byte instruction queue and
b) 8088 has 8-bit data bus.

The function of each block is the same as in 8086.

The addressing capability of 8088 is 1-Mbyte; therefore, it needs 20 address bits, i.e. 20 addressing lines. While handling this 20-bit address, the segmented memory scheme is used and the complete physical address forming procedure is the same as explained in case of 8086. While physically interfacing memory to 8088, there is nothing like an even address bank or odd address bank. The complete memory is homogeneously addressed as a bank of 1-Mbyte memory locations using the segmented memory scheme. This change in hardware is completely transparent to software. As a result of the modified data bus, the 8088 can access only a byte at a time. This fact reduces the speed of operation of 8088 as compared to 8086, but the 8088 can process the 16-bit data internally. On account of this change in bus structure, the 8088 has slightly different timing diagrams than 8086.

1.8. Comparison between 8086 and 8088

The 8088, with an 8-bit external data bus, has been designed for internal 16-bit processing capability. Nearly all the internal functions of 8088 are identical to 8086. The 8088 uses the external bus in the same way as 8086, but only 8 bits of external data are accessed at a time. While fetching or writing the 16-bit data, the task is performed in two consecutive bus cycles. As far as the software is concerned, the chips are identical, except in case of timings. The 8088, thus may take more time for execution of a particular task as compared to 8086.

All the changes in 8088 over 8086 are directly or indirectly related to the 8-bit, 8085 compatible data and control bus interface.

1. The pre-decoded code queue length is reduced to 4 bytes in 8088, whereas the 8086 queue contains 6 bytes. This was done to avoid the unnecessary prefetch operations and optimize the use of the bus by BIU while prefetching the instructions.
2. The 8088-bus interface unit will fetch a byte from memory to load the queue each time, if at least 1 byte is free. In case of 8086, at least 2 bytes should be free for the next fetch operation.

3. The 8-bit external data bus affects the overall execution time of the instructions in 8088. All the 16-bit operations now require additional 4 clock cycles. The CPU speed is also limited by the speed of instruction fetches.

The pin assignments of both the CPUs are nearly identical, however, they have the following functional changes.

1. $A_8 - A_{15}$ already latched, all time valid address bus.
2. $BHE$ has no meaning as the data bus is of 8-bits only.
3. $SS_0$ provides the $S_0$ status information in minimum mode.
4. $IO/M$ has been inverted to be compatible with 8085 bus structure.

1.9. The Minimum and Maximum Modes

The logic level applied to the MN/$\overline{MX}$ input, pin 33, determines the operating mode of the 8086. If pin 33 is asserted high, then the 8086 will function in minimum mode, and pins 24 through 31 will have the functions shown in parentheses next to the pins in the pin diagram. In the minimum mode, for example, pin 29 will function as $\overline{WR}$, which will go low any time the 8086 writes to a port or to a memory location. The $RD$, $WR$, and $M/IO$ signals form the heart of the control bus for a minimum mode 8086 system. The 8086 is operated in minimum mode in systems such as the SDK-86/EV-Z2 where it is the only microprocessor on the system buses.

If the MN/$\overline{MX}$ pin is asserted low, then the 8086 is in maximum mode. In this mode, pins 24 through 31 will have the functions described by the mnemonics next to the pins in the pin diagram. In this mode, the control bus signals ($S_0$, $S_1$, and $S_2$) are sent out in encoded form on pins 26, 27, and 28. An external bus controller device decodes these signals to produce the control bus
signals required for a system, which has two or more microprocessors sharing the same buses.

1.10. WHAT IS THE NEED FOR SEGMENTATION?

At this point you may be wondering why Intel designed the 8086 family devices to access memory using the segment: offset approach rather than accessing memory directly with 20-bit addresses. The segment: offset, scheme requires only a 16-bit number to represent the base address for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.

The second reason for segmentation has to do with the type of microcomputer in which an 8086-family CPU is likely to be used. In a timesharing system, several users share a CPU. The CPU works on one user's program for perhaps 20 ms, then works on the next user's program for 20 ms. After working 20 ms for each of the other users, the CPU comes back to the first user's program again. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy. Each user's program can be assigned a separate set of logical segments for its code and data. The user's program will contain offsets or displacements from these segment bases. To change from one user's program to a second user's program, all that the CPU has to do is to reload the four segment registers with the segment base addresses assigned to the second user's program. In other words, segmentation makes it easy to keep users’ programs and data separate from one another, and segmentation makes it easy to switch from one user's program to another user's program.