The second option for improving the processing power of the CPU has to do with the internal working of the CPU. In the 8085 microprocessor, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time as shown in Figure 1-2. It is important to point out that Figure 1.2 is not meant to imply that the amount of time for fetch and execute are equal.

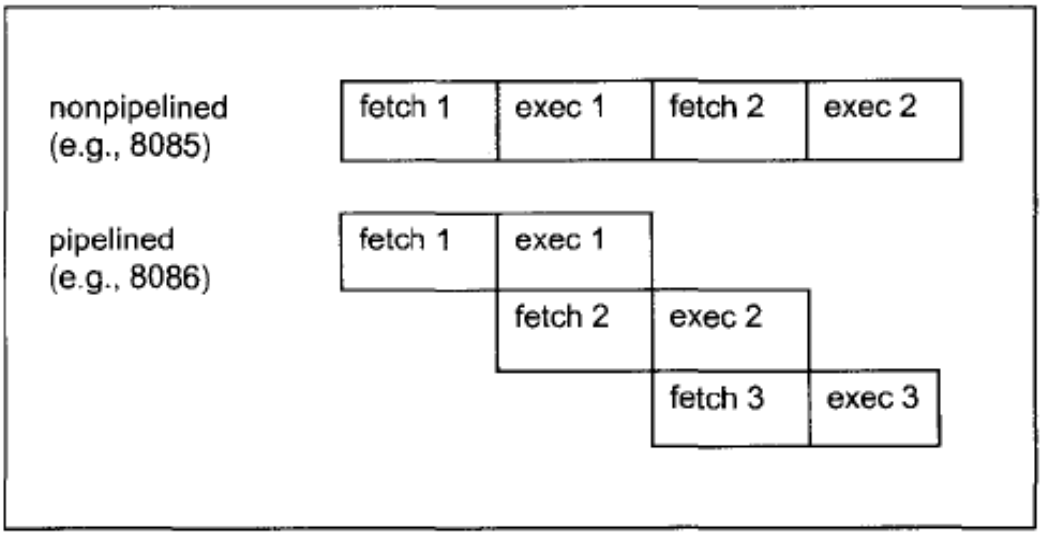


Figure (1.2) Pipelined vs. non pipelined Execution

Intel implemented the concept of pipelining in the 8088/86 by splitting the internal structure of the microprocessor into two sections: the **execution unit (EU)** and the **bus interface unit (BIU)**.

These two sections work simultaneously. The BIU accesses memory and peripherals while the EU executes instructions previously fetched.

This works only if the BIU keeps ahead of the EU; thus the BIU of the 8088/86 has a buffer, or queue (see Figure 1.1). The buffer is 4 bytes long in the 8088 and 6 bytes in the 8086. If any instruction takes too long to execute, the queue is filled to its maximum capacity and the buses will sit idle. The BIU fetches a new instruction whenever the queue has room for 2 bytes in the 6-byte 8086 queue, and for 1 byte in the 4-byte 8088 queue. In some circumstances, the microprocessor must flush out the queue. For example, when a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information in the queue that was fetched previously is discarded. In this situation the EU must wait until the BIU fetches the new instruction. This is referred to in computer

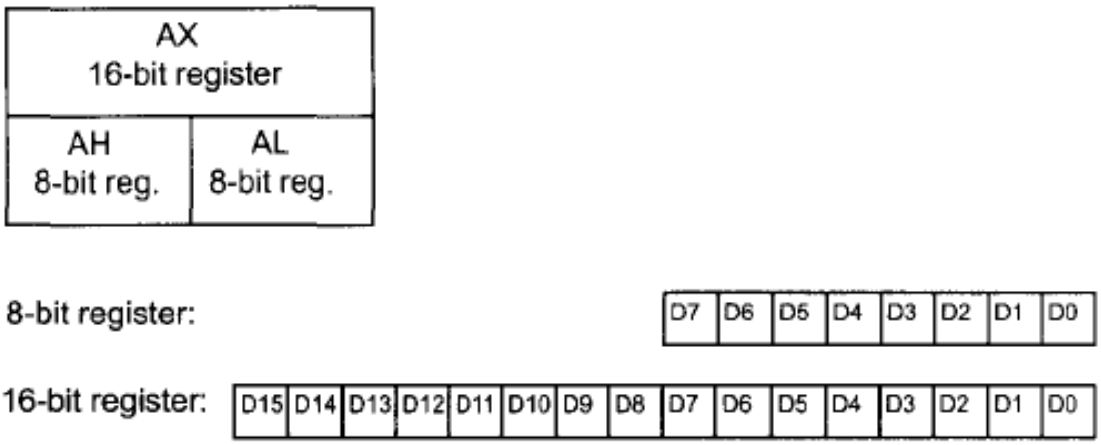
science terminology as a branch penalty. In a pipelined CPU, this means that too much jumping around reduces the efficiency of a program.

Pipelining in the 8088/86 has two stages: fetch and execute, but in more powerful computers pipelining can have many stages. The concept of pipelining combined with an increased number of data bus pins has, in recent years, led to the design of very powerful microprocessors.

Registers

In the CPU, registers are used to store information temporarily. That information could be one or two bytes of data to be processed or the address of data. The registers of the 8088/86 fall into the six categories outlined in Table 1.2.

The general-purpose registers in 8088/86 microprocessors can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In the 8088/86, data types are either 8 or 16 bits. To access 12-bit data, for example, a 16-bit register must be used with the highest 4 bits set to 0. The bits of a register are numbered in descending order, as shown below.



Different registers in the 8088/86 are used for different functions, and since some instructions use only specific registers to perform their tasks, the use of registers will be described in the context of instructions and their application in a given program. The first letter of each general register indicates its use. AX is used for the accumulator, BX as a base addressing register, CX is used as a counter in loop operations, and DX is used to point to data in I/O operations.

Table 1-2: Registers of the 8086/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note:

The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

1.3 Assembly language programming

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The **operands** are the data items being manipulated, and the **mnemonics** are the commands to the CPU, telling it what to do with those items.

We introduce Assembly language programming with two widely used instructions: the ***move*** and ***add*** instructions.

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

MOV destination, source ; copy source operand to destination

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand.

For example, the instruction "MOV DX,CX" copies the contents of register CX to register DX. After this instruction is executed, register DX will have the same value as register CX. The MOV instruction does not affect the source operand.