Input-Output Organization

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user.

Peripheral Devices

Input or output devices attached to the computer are also called peripherals.

- The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.
- Printers provide a permanent record on paper of computer output data.
- Magnetic tapes are used mostly for storing files of data.
- Magnetic disks have high-speed rotational surfaces coated with magnetic material.

Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

A typical communication link between the processor and several peripherals is shown in Fig.36. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

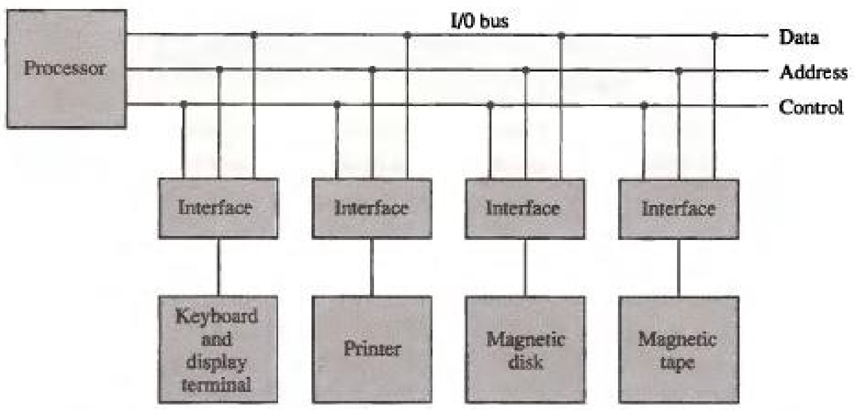
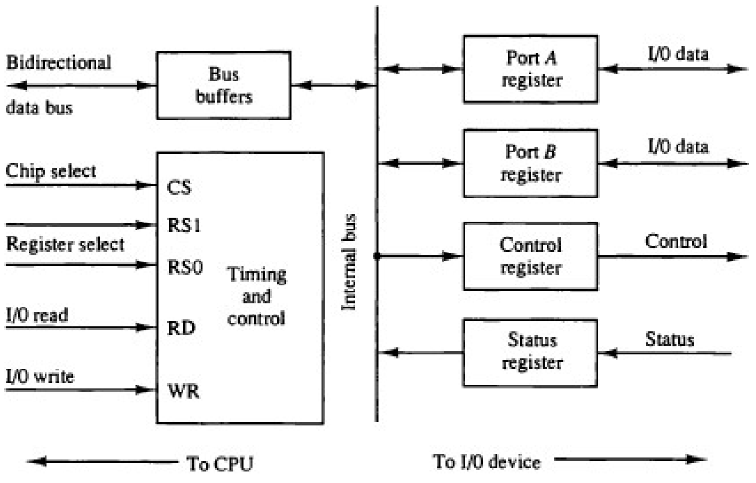


Figure 36 Connection of I/O bus to input-output devices.

Isolated I/O versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure 37 Example of I/O interface unit.

This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. In a memory-mapped I/O organization there is no specific input or output instructions. Computers with memory-mapped I/O can use memory-type instructions to access I/O data.

An example of an I/O interface unit is shown in block diagram form in Fig.37. It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. Two way of achieving this:

- The strobe: pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- The handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers.

The strobe may be activated by either the source or the destination unit. Figure 38 shows a source-initiated transfer and the timing diagram.

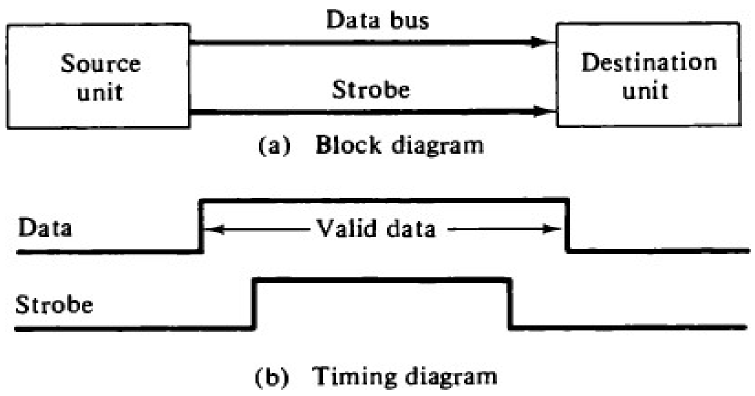


Figure 38 Source-initiated strobe for data transfer.

Fig.39 shows the strobe of a memory-read control signal from the CPU to a memory.

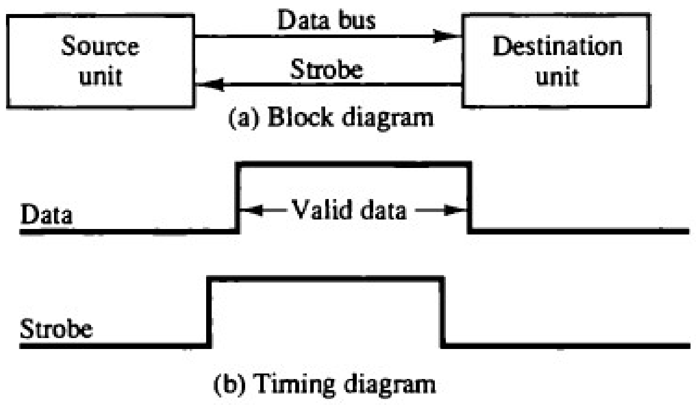


Figure 39 Destination-initiated strobe for data transfer.

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that two-wire control initiates the transfer.

Figure 40 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. Figure 41 the destination-initiated transfer using handshaking lines. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.

Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an w-bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors. Serial transmission can be synchronous or asynchronous. A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always(0).
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

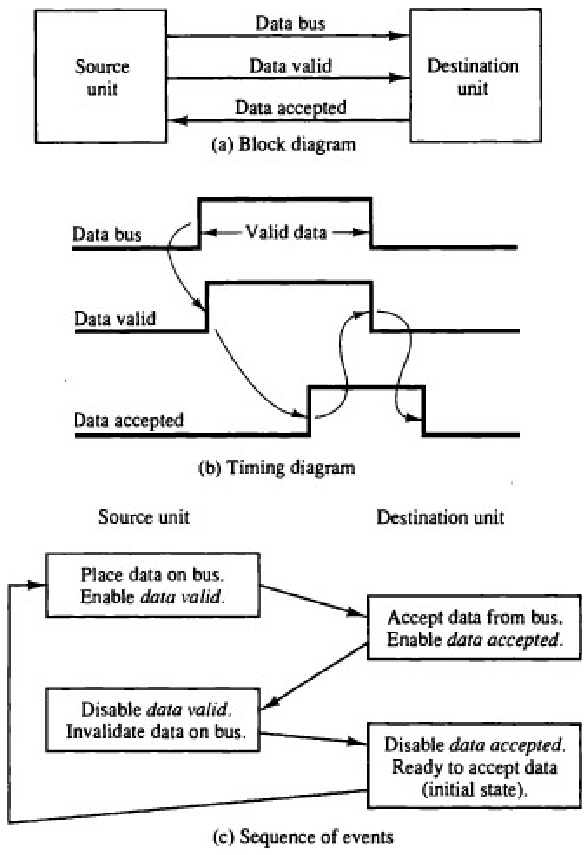

Source unit
Destination unit
Data bus
Data valid
Data accepted
(a) Block diagram
Data bus
Valid data
Data valid
Data accepted
(b) Timing diagram
Source unit
Destination unit
Place data on bus. Enable *data valid*.
Accept data from bus. Enable *data accepted*.
Disable *data valid*. Invalidate data on bus.
Disable *data accepted*. Ready to accept data (initial state).
(c) Sequence of events

Figure 40 Source-initiated transfer using handshaking.

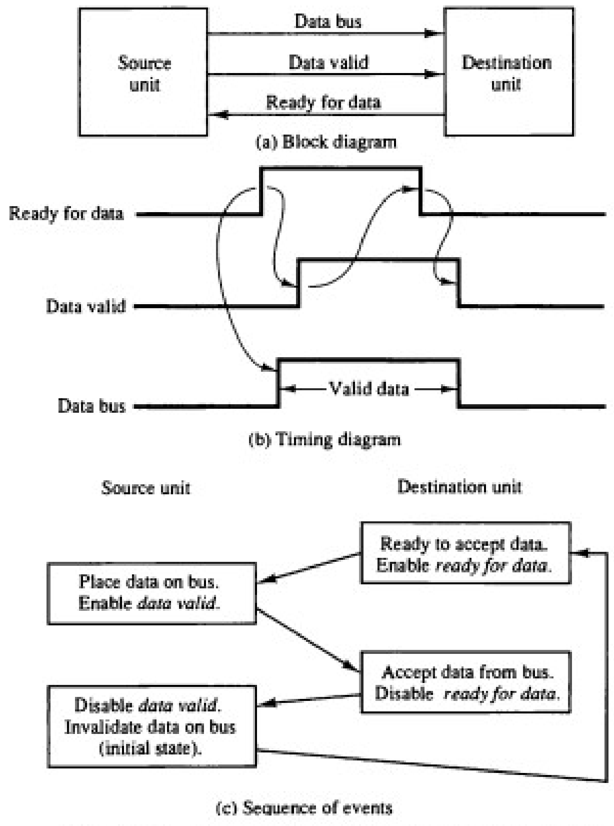

Source unit
Destination unit
Data bus
Data valid
Ready for data
(a) Block diagram
Ready for data
Data valid
Data bus
Valid data
(b) Timing diagram
Source unit
Destination unit
Ready to accept data. Enable *ready for data*.
Place data on bus. Enable *data valid*.
Accept data from bus. Disable *ready for data*.
Disable *data valid*. Invalidate data on bus (initial state).
(c) Sequence of events

Figure 41 Destination-initiated transfer using handshaking.

An example of serial transmission format is shown in Fig. 42.

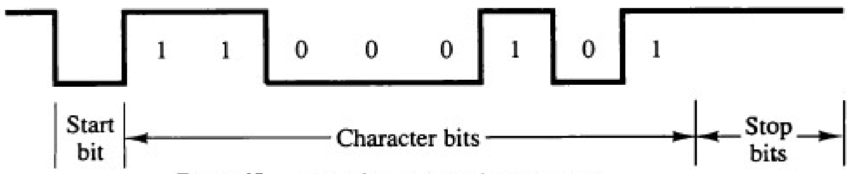


Figure 42 Asynchronous serial transmission.

As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format have a transfer rate of 110 baud. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver-transmitter (UART).

Modes of Transfer

Data transfer between the central computer and I/O devices may be handled in a variety of modes. three possible modes:

1. Programmed I/O: The operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. The CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 43.

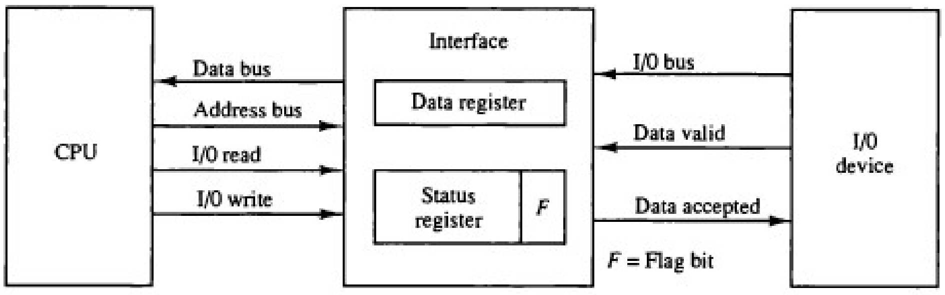


Figure 43 Data transfer from I/O device to CPU.

2. Interrupt-initiated I/O: It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. This method of connection between three devices and the CPU is shown in Fig. 44.

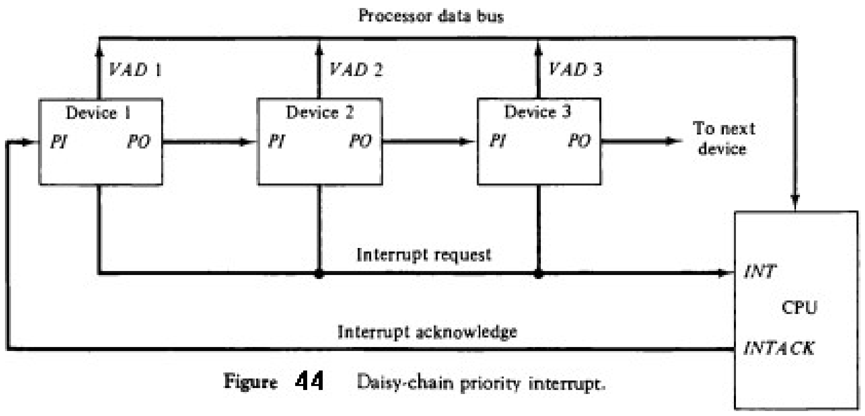


Figure 44 Daisy-chain priority interrupt.

3. Direct memory access (DMA): the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. This method of connection between devices and the memory is shown in Fig. 45.

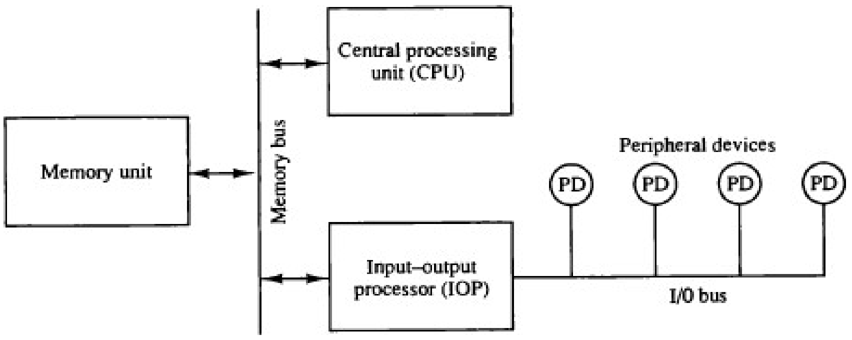


Figure 45 Block diagram of a computer with I/O processor.

Pipelining

Pipelining is a technique of decomposing a sequential process into sub-operations; with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows.

General Considerations

Any operation that can be decomposed into a sequence of sub-operations of about the same complexity can be implemented by a pipeline processor. The general structure of a four-segment pipeline is illustrated in Fig. 46. The operands pass through all four segments in a fixed sequence.

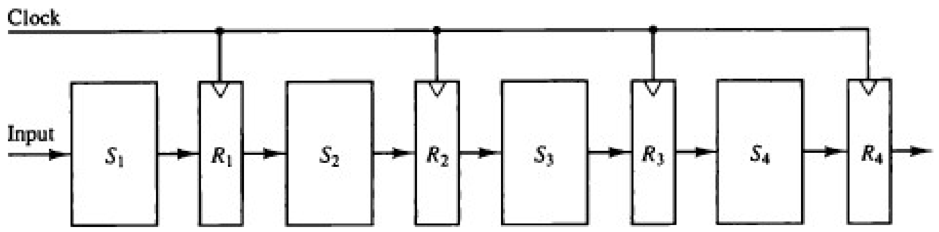
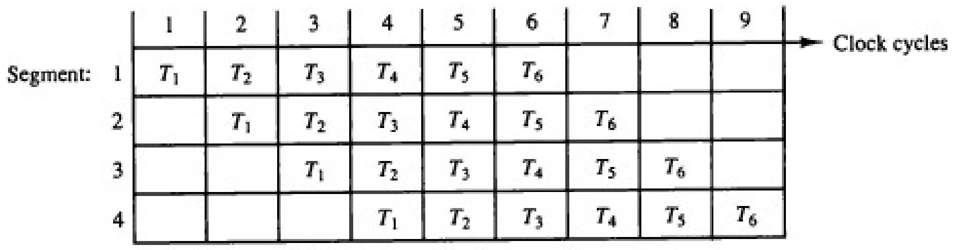


Figure 46 Four-segment pipeline.

The space-time diagram of a four-segment pipeline is demonstrated in Fig47.



Segment:	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Clock cycles

Figui 47 Space-time diagram for pipeline.

The speedup(S) of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio: $S = \frac{nt_n}{(k+n-1)t_p}$

As the number of tasks increases, n becomes much larger than $k-1$, and $k+n-1$ approaches the value of n. Under this condition, the speedup becomes:

$$S = \frac{t_n}{t_p}$$

numerical example: Let the time it takes to process a sub-operation in each segment be equal to t_p= 20 ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take

$$(k+n-1)t_p = (4+99) \times 20 = 2060ns$$

to complete. Assuming that t = ktp = 4 x 20 = 80 ns,

a non-pipeline system requires:

$$nkt_p = 100 \times 80 = 8000ns$$

to complete the 100 tasks. The speedup ratio is equal to:

$$^{8000}/_{2060} = 3.88$$

Instruction Pipeline

The computer needs to process each instruction with the following sequence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Figure 48 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

The four segments are represented in the flowchart:

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

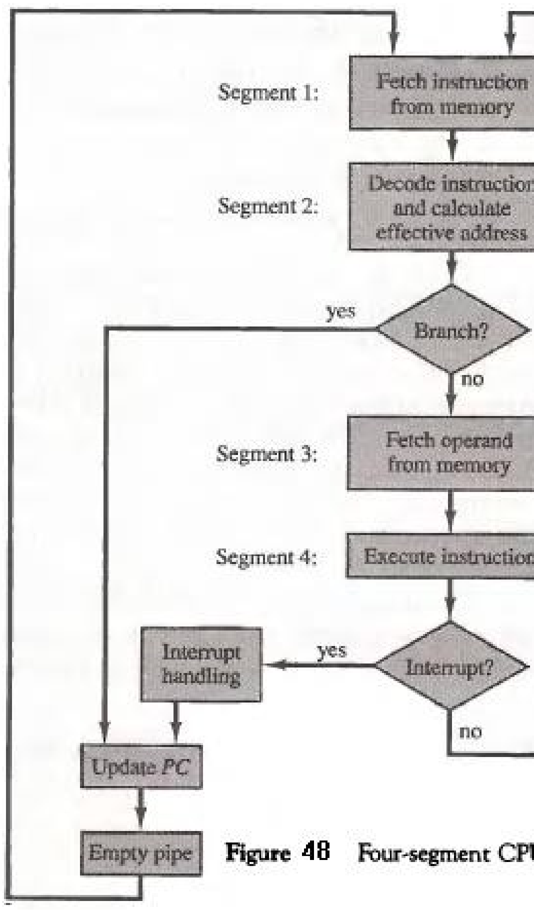


Figure 48 Four-segment CPU pipeline.

A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units.

<u>Instruction-Level Parallelism</u>

Contrary to pipeline techniques, instruction-level parallelism (ILP) is based on the idea of multiple issue processors (MIP). An MIP has multiple pipelined datapaths for instruction execution. Each of these pipelines can issue and execute one instruction per cycle. Figure 49 shows the case of a processor having three pipes. For comparison purposes, we also show in the same figure the sequential and the single pipeline case.

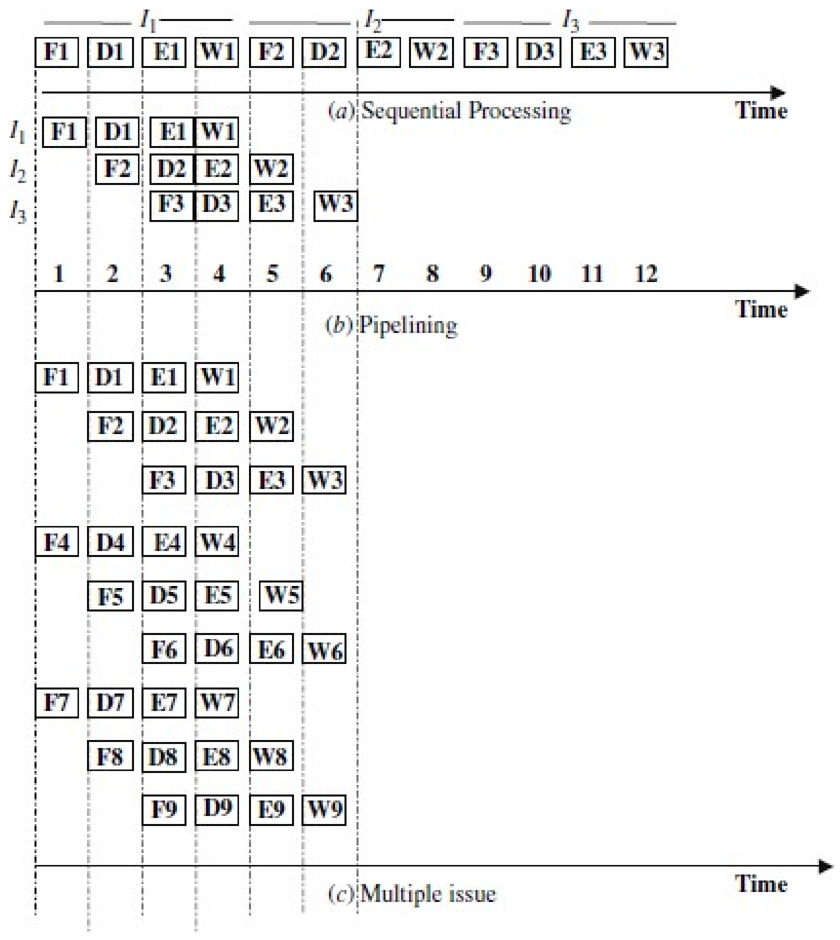


Figure 49 Multiple issue versus pipelining versus sequential processing

Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

an example of a pipeline unit for floating-point addition and subtraction. The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

A, B are two fractions that represent the mantissas and a, b are the exponents. The sub-operations that are performed in the four segments are:

1. Compare the exponents.

2. Align the mantissas.

3. Add or subtract the mantissas.

4. Normalize the result.

Numerical example may clarify the sub-operations performed in each segment. For simplicity, we use decimal numbers, although Fig.49 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $(3 - 2 = 1)$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain:

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum:

$$Z = 1.0324 \times 10^3$$

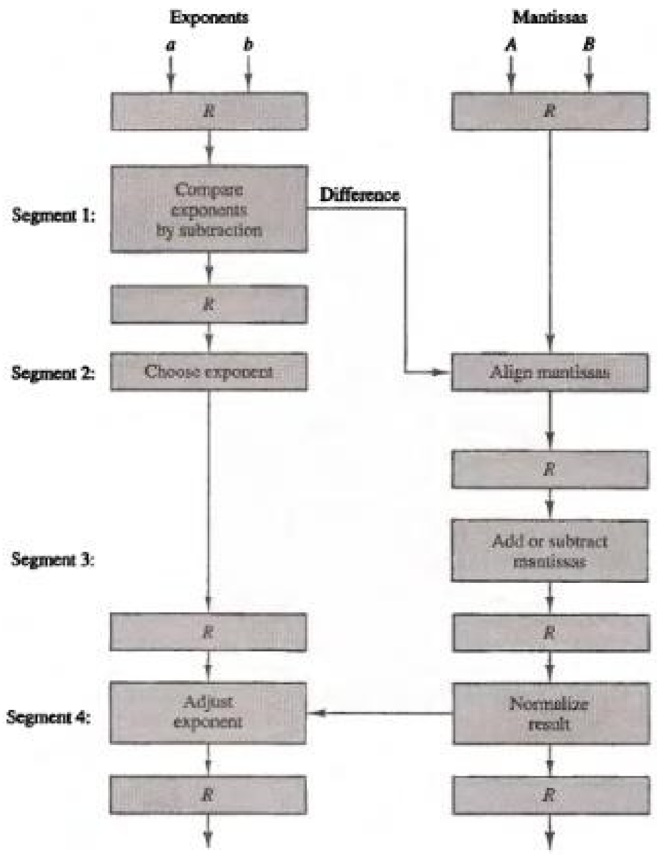


Figure 49 Pipeline for floating-point addition and subtraction.

Suppose that the time delays of the four segments are $t_1 = 60ns, t_2 = 70ns, t_3 = 100ns$, $t_4 = 80ns$, and the interface registers have a delay of $t_r = 10ns$. The clock cycle is chosen to be $t_p = t_3 + t_r = 110ns$. An equivalent non-pipeline floating point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320ns$. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the non-pipelined adder.

Supercomputers

Supercomputers are very powerful, high-performance machines used mostly for scientific computations. To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel. Supercomputers also use special techniques for removing the heat from circuits to prevent them from burning up because of their close proximity.

A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.

Delayed Branch

Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M$[address 1]
2. LOAD: $R2 \leftarrow M$[address 2]
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: M[address 3] $\leftarrow R3$

If the three-segment pipeline proceeds: (I: Instruction fetch, A:ALU operation, and E: Execute instruction) without interruptions, there will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment. This can be seen from the timing of the pipeline shown in Fig. 50(a). The E segment in clock cycle 4 is in a process of placing the memory data into R2. The A segment in clock cycle 4 is using the data from R2, but the value in R2 will not be the correct value since it has not yet been transferred from memory. It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. It was shown in Fig. 50 that a branch instruction delays the pipeline operation by NOP instruction until the instruction at the branch address is fetched.

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Figure 50 Three-segment pipeline timing.