Chapter 19

Input/Output Organization

19–1 There are two main reasons for using an I/O controller. First, different I/O devices exhibit different characteristics and, if these devices were connected directly, the CPU would have to understand and respond appropriately to each I/O device. This would cause the CPU to spend a lot of time interacting with I/O devices and spend less time executing user programs. If we use an I/O controller, this controller could provide the necessary low-level commands and data for proper operation of the associated I/O device. Often, for complex I/O devices such as disk drives, there are special I/O controller chips available.

The second reason for using an I/O controller is that the amount of electrical power used to send signals on the system bus is very low. This means that the cable connecting the I/O device has to be very short (a few centimeters at most). I/O controllers typically contain driver hardware to send current over long cables that connect I/O devices.

19–2 Memory-mapped I/O maps I/O port addresses to memory address space. Processors such as the PowerPC and MIPS support only memory-mapped I/O. In these systems, writing to an I/O port is similar to writing to a memory location. Memory-mapped I/O does not require any special consideration from the processor. Thus, all processors inherently support memory-mapped I/O.

Isolated I/O maintains an *I/O address space* that is separate from the memory address space. The Pentium supports isolated I/O. In these systems, special I/O instructions are needed to access the I/O address space. The Pentium provides two basic I/O instructions—`in` and `out`—to access I/O ports.

19–3 Memory-mapped I/O maps I/O port addresses to memory address space. Memory-mapped I/O does not require any special consideration from the processor. However, it takes part of the memory address space (MAS) for I/O mapping. On the other hand, isolated I/O maintains an I/O address space that is separate from the memory address space. Thus, the complete MAS can be used for memory. The disadvantage of isolated I/O is that they require special I/O instructions to access the I/O address space.

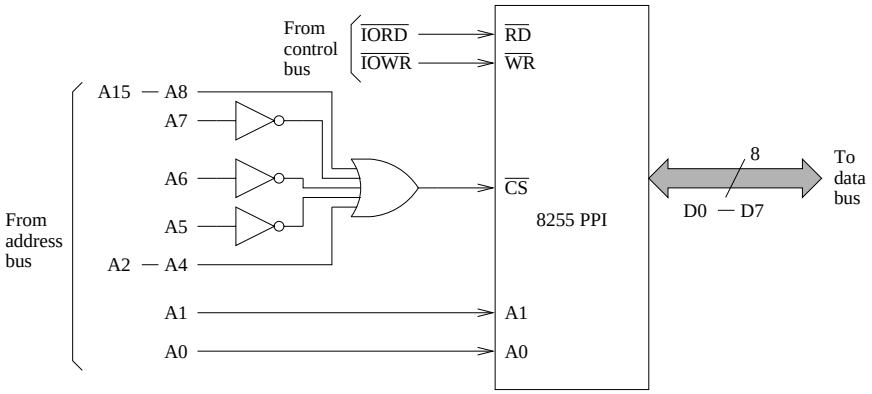
19–4 No. All systems support memory-mapped I/O simply because there is no extra support required. Systems designed with processors supporting the isolated I/O have the flexibility of using either the memory-mapped I/O or isolated I/O.

19–5 Memory-mapped I/O maps I/O port addresses to memory address space. As a consequence, it can use the standard memory read/write instructions to access I/O devices. There is no need for special I/O instructions.

19–6 The scan code of a key does not have any relation to the ASCII value of the corresponding character. The scan codes are assigned based where the key is located on the keyboard.

19–7 The direct addressing mode can only be used to access the first 256 ports. In this case, the I/O port address, which is in the range 0 to FFH, is given directly in the I/O instruction. In the indirect form, the I/O port address is given indirectly via the DX register. The contents of the DX register are treated as the port address. This form can be used to access any I/O port.

19–8 The modified design is shown below:



19–9 No, it is not possible to the four I/O ports to 62H to 65H. The reason is that the least significant two address lines (A1 and A0) are used to select a port, which means that the first port address must have zeroes for these two bits. Since 62H implies that A1 = 1, we cannot map the I/O ports to these four addresses.

19–10 Programmed I/O involves the processor in the I/O data transfer. The processor repeatedly checks to see if a particular condition is true. Typically, it busy-waits until the condition is true. Thus, programmed I/O mechanism wastes processor time. However, it does not require any hardware/system support to implement. Direct memory access, on the other hand, relieves the processor of the low-level data transfer chore. However, DMA needs hardware support as it is implemented by using a DMA controller.

19–11 The temporary register is used to hold the data during memory-to-memory transfer of data.

19–12 The following simple rule can used to remember the bit positions tested by each parity bit:

Parity bit P_1 checks all those bits whose bit position, when expressed in binary, consists of 1 in the least significant bit position (i.e., in 2^0 position).

Parity bit P_2 checks all those bits whose bit position, when expressed in binary, consists of 1 in the second least significant bit position (i.e., in 2^1 position).

From this description, we can generalize as follows: In general, P_{2^k} checks a bit whose bit position has a 1 in the 2^k position.

Check bit P_1 looks at bit positions 1, 3, 5, 7, 9, 11, and 13. Note that all these bit positions have 1 in their least significant bit (i.e., 2^0 bit position).

Check bit P_4 looks at bit positions 4, 5, 6, 7, 12, 13, and 14. Note that all these bit positions have 1s in their 2^2 bit positions (i.e., in the third bit position from right).

This verifies that our simple scheme works.

19–13 The clue comes from the following statement: The error bit position is the sum of the error parity bits with P_1 counted as 1, P_2 counted as 2, P_4 counted as 4, and so on.

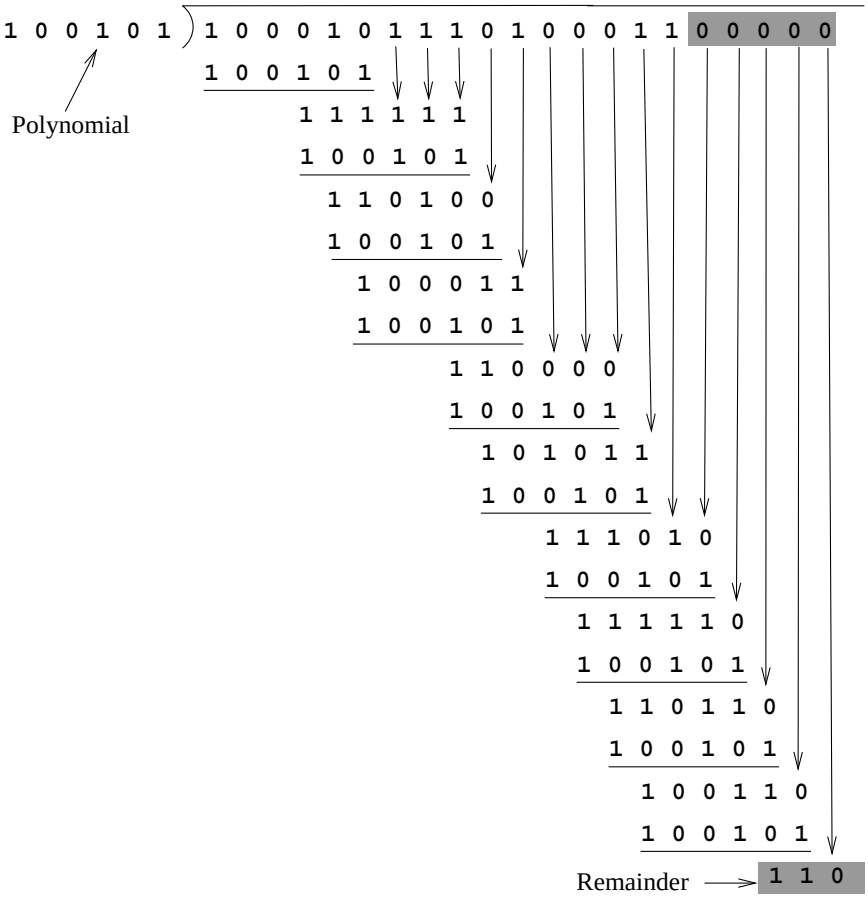
If we have n bits, the number of parity bits p should be such that the p-bit number should be able to point to any one of the n bits. Thus, $2^p \geq n + 1$ and $2^{p-1} < n + 1$, which leads to $p = \lceil log_2(n + 1) \rceil$.

19–14 The overhead is shown in the following table:

Data bits (n)	Parity bits (p)	Overhead (%)
1	1	$\frac{1}{1} \times 100 = 100.00$
2	2	$\frac{2}{2} \times 100 = 100.00$
3	2	$\frac{2}{3} \times 100 = 67.78$
4	3	$\frac{3}{4} \times 100 = 75.00$
5	3	$\frac{3}{5} \times 100 = 60.00$
6	3	$\frac{3}{6} \times 100 = 50.00$
7	3	$\frac{3}{7} \times 100 = 42.85$
8	4	$\frac{4}{8} \times 100 = 50.00$
9	4	$\frac{4}{9} \times 100 = 44.44$
10	4	$\frac{4}{10} \times 100 = 40.00$
11	4	$\frac{4}{11} \times 100 = 36.36$
12	4	$\frac{4}{12} \times 100 = 33.33$
13	4	$\frac{4}{13} \times 100 = 39.77$
14	4	$\frac{4}{14} \times 100 = 28.57$
15	4	$\frac{4}{15} \times 100 = 26.67$
16	5	$\frac{5}{16} \times 100 = 31.25$
17	5	$\frac{5}{17} \times 100 = 29.41$
18	5	$\frac{5}{18} \times 100 = 27.78$
19	5	$\frac{5}{19} \times 100 = 26.32$
20	5	$\frac{5}{20} \times 100 = 25.00$

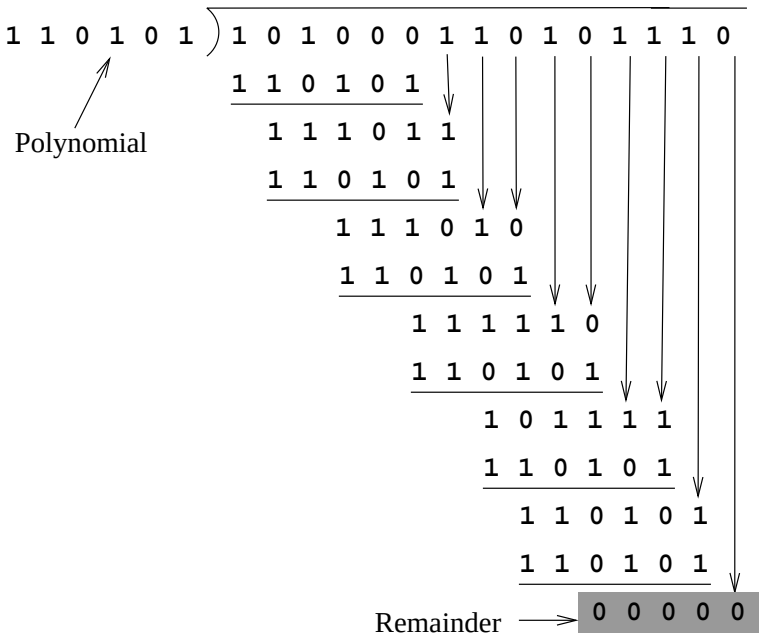
In the above table, we used the following formula to compute p: $p = \lceil log_2(n+1) \rceil$. From this data we observe that the overhead decreases as we increase n. When we plot this, we observe a "saw-tooth" shape, with the overhead jumping whenever the number of parity bits increases by one (for example, when n changes from 7 to 8, or from 15 to 16, and so on).

19–15 The CRC calculation is shown below:

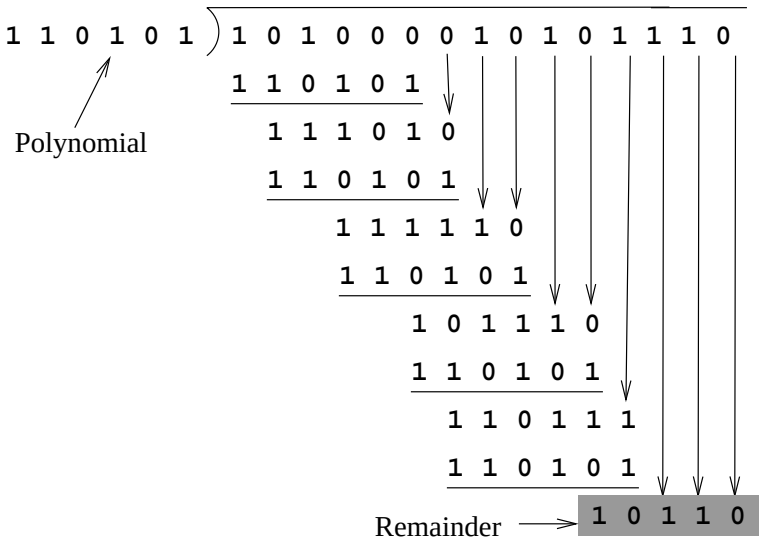


The codeword is: 1000101110100011 00110

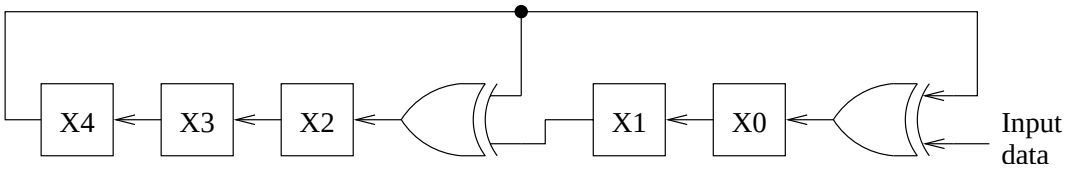
19–16 As shown below, the remainder is zero. This indicates that the codeword has been received correctly.



19–17 As shown below, the remainder is not zero. This indicates that the codeword has been received incorrectly.



19–18 The circuit is shown below:

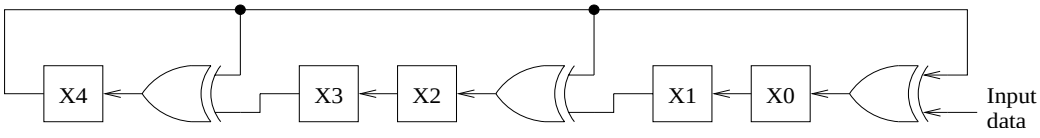


A trace of this circuit is shown below:

Clock	X4	X3	X2	X1	X0	Next data bit
0	0	0	0	0	0	1
1	0	0	0	0	1	0
2	0	0	0	1	0	1
3	0	0	1	0	1	0
4	0	1	0	1	0	0
5	1	0	1	0	0	1
6	0	1	1	0	0	0
7	1	1	0	0	0	1
8	1	0	1	0	0	0
9	0	1	1	0	1	0
10	1	1	0	1	0	0
11	1	0	0	0	1	0
12	0	0	1	1	1	0
13	0	1	1	1	0	—

As shown in this trace, the remainder is 01110.

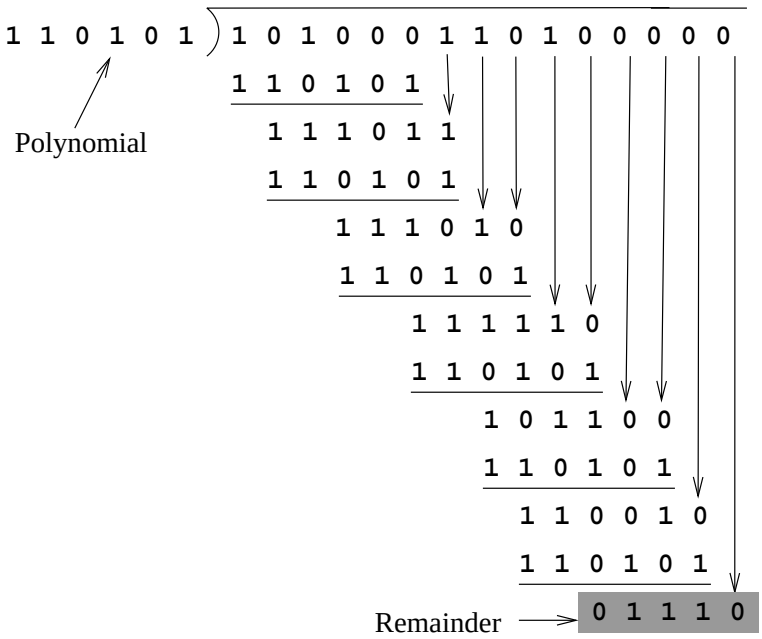
19–19 The circuit is shown below:



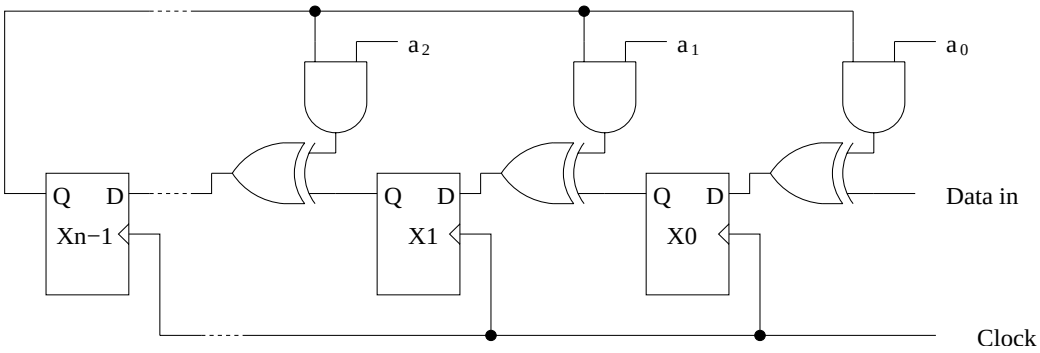
A trace of this circuit, given below, shows that the remainder is 01110.

Clock	X4	X3	X2	X1	X0	Next data bit
0	0	0	0	0	0	1
1	0	0	0	0	1	0
2	0	0	0	1	0	1
3	0	0	1	0	1	0
4	0	1	0	1	0	0
5	1	0	1	0	0	0
6	1	1	1	0	1	1
7	0	1	1	1	0	1
8	1	1	1	0	1	0
9	0	1	1	1	1	1
10	1	1	1	1	1	0
11	0	1	0	1	1	0
12	1	0	1	1	0	0
13	1	1	0	0	1	0
14	0	0	1	1	1	0
15	0	1	1	1	0	—

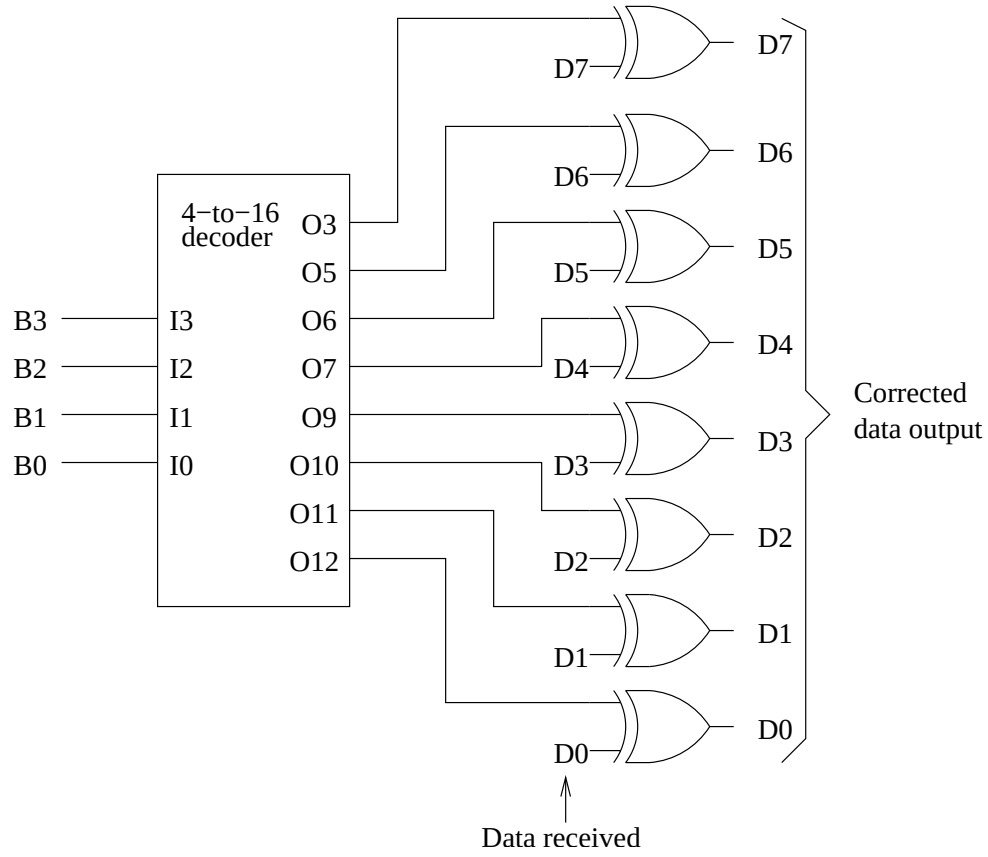
As shown below, the remainder is indeed 01110:



19–20 The circuit is shown below:



19–21 The circuit is shown below:



Note that the circuit leaves out some unused outputs from the decoder.

19–22 When transmitting a character, the start bit pulls the communication line low. This alerts the receiver that a byte is coming. It also identifies the bit boundary. Since the receiver knows the bit period, it samples the transmission line in the middle of the bit cell.

19–23 The stop bit serves two purposes:

- Imagine what happens if we don't have a stop bit. Suppose the most significant bit of the byte is 0. Then, unless we force some idle time on the line, the start bit of the next byte will not cause the transition to identify the start of a new byte. The stop bit forces the line to go high between byte transmissions.
- Stop bits also give breathing time for the receiver to assemble the byte and hand it over to the receiver system before monitoring the line for the start bit. Typically, systems can use 1, 1 $\frac{1}{2}$, or 2 stop bits.

19–24 Before the USB and IEEE 1394, computer users faced several problems when attaching peripherals. Here we list some of these problems.

Device-Specific Interfaces: PCs tended to have various device-specific interfaces. Some example connectors we will find on a PC include the PS/2, serial, parallel, monitor, microphone, speakers, modem, SCSI, and Ethernet. In most cases, each connection uses its own connector and cable, leading to cable clutter. In contrast to this scenario, USB uses a single connector type to connect any device.

Nonshareable Interfaces: Standard interfaces support only one device. For example, we can connect only one printer to the parallel interface. In contrast, the USB supports up to 127 devices per USB connection. For example, we can connect a keyboard, a mouse, and speakers to a single USB port using a single cable type.

I/O Address Space and Interrupt Request Problems: Adding a new peripheral device often causes I/O address and interrupt request (IRQ) conflicts. One may end up spending countless hours in debugging the conflict problem. In contrast, the USB does not require memory or address space. There is also no need for interrupt request lines.

Installation and Configuration: Using the standard interfaces, adding a new peripheral device is often a time-consuming and frustrating experience for novice users. It may often involve opening the box and installing expansion cards and configuring jumpers or DIP switches. In contrast, the USB and IEEE 1394 support true plug-and-play connectivity. It avoids unpleasant tasks such as setting jumpers and configuring the new device.

No Hot Attachment: We are too familiar with the dreaded sequence of restarts whenever we attach a new device. Attaching USB and IEEE 1394 devices is easy: we don't have to turn off the computer and restart after installing the new device. We can hot plug the device and the system will automatically detect the device and configure it for immediate use.

19–25 In NRZ encoding, a 0 is represented by a low level and a 1 by a high level. Even though this scheme is simple to implement, it has two serious problems:

- Signal transitions do not occur if we are transmitting long strings of zeros or ones. Signal transitions are important for the receiver to recover data.
- In a noisy medium, it is difficult to detect zero- and one-levels. It is far easier to detect a transition, either from 0 to 1 or 1 to 0.

NRZI encoding solves the two main problems associated with NRZ encoding. In NRZI encoding, signal level does not play any role. It only looks for signal transitions. Thus, it improves reliability of data transmission. Furthermore, it also solves the long strings of zeros. A long string of zeros forces the NRZI signal to alternate. However, it does not solve the problem with long strings of ones.

19–26 The USB does not support interrupts in the traditional sense. Instead, the USB uses polling, which is similar to the busy-waiting used by the programmed I/O. To get acceptable performance,

we have to select an appropriate polling frequency. If the frequency is too high, we will waste the bandwidth. On the other hand, if we use too low a frequency, we may lose data. The frequency of USB interrupt transfers can be adjusted to meet the device requirements. In the USB 1.1, the polling interval can range from 1 to 255 ms, that is, from 1000 times to about 4 times a second. The USB uses an endpoint descriptor to specify the polling interval, in increments of 1 ms.

19–27 The main difference between the UHC and OHC controllers is the policy used to schedule the four types of transfers (interrupt, isochronous, control, and bulk transfers). Both controllers, however, use 1 ms frames to schedule the transfers.

The UHC schedules periodic transfers—isochronous and interrupt—first. These transfers are followed by the control and bulk transfers. The periodic transfers can take up to 90% of the bandwidth and the control transfers are allocated a guaranteed 10% bandwidth. Bulk transfers are scheduled only if there is bandwidth available after scheduling the other three transfers.

The OHC uses a slightly different scheduling policy. It reserves space for nonperiodic transfers (control and bulk) at the beginning of the frame such that these transfers are guaranteed 10% of the bandwidth. Next periodic transfers are scheduled to guarantee 90% of the bandwidth. If there is time left in the frame, nonperiodic transfers are scheduled.

19–28 A bus-powered hub does not require an extra power supply; the hub uses the power supplied by the USB. Bus-powered hubs can be connected to an upstream port that can supply a full 500 mA current. The downstream ports of the hub can only provide 100 mA of current. Furthermore, the number of ports is limited to four. For this reason, seven-port USB hubs are not bus-powered.

19–29 The USB is processor-centric (i.e., the processor initiates the transfers). The USB typically supports host-to-peripheral applications. The IEEE 1394, on the other hand, supports peer-to-peer communication without involving the processor. Since IEEE 1394 supports peer-to-peer communication, we need a bus arbitration mechanism.

19–30 Typically the root node serves as the IRM. This determination is done as part of the self-identification phase that configures the nodes. During this phase, self-id packets carry information about the capability of a node to become the IRM. If there is more than one contender for the IRM, the node with the highest physical id gets to be the IRM. Since the root node has the highest id, it often acts as the IRM for the network.