Problem 3

\[
\begin{array}{ccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 \\
+ & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

One's complement = 11101110.

To detect errors, the receiver adds the four words (the three original words and the checksum).

The answer should be 11111111

If the sum contains a zero, the receiver knows there has been an error.

All one-bit errors will be detected, but two-bit errors can be undetected. For example, say, if the last digit of the first word is converted to a 0 and the last digit of the second word is converted to a 1, the answer would remain as 11111111.

Problem 4

(a) Adding the two bytes gives 10011101. Taking the one’s complement gives 01100010
(b) Adding the two bytes gives 00011110; the one’s complement gives 11100001.
(c) first byte = 00110101 ; second byte = 01101000.

Problem 5

No, the receiver cannot be absolutely certain that no bit errors have occurred. This is because of the manner in which the checksum for the packet is calculated. If the corresponding bits (that would be added together) of two 16-bit words in the packet were 0 and 1 then even if these get flipped to 1 and 0 respectively, the sum still remains the same. Hence, the 1s complement the receiver calculates will also be the same. This means the checksum will verify even if there was transmission error.

It should be remembered that the probability of two bits flipping at the same corresponding position is extremely low. However, there is always a non-zero probability of such occurrences.

Problem 7

To best answer this question, consider why we needed sequence numbers in the first place. We saw that the sender needs sequence numbers so that the receiver can tell if a data packet is a duplicate of an already received data packet. In the case of ACKs, the sender does not need this info (i.e., a sequence number on an ACK) to tell detect a duplicate ACK.
A duplicate ACK is obvious to the rdt3.0 sender, since when it has received the original ACK it transitioned to the next state. The duplicate ACK is not the ACK that the sender needs and hence is ignored by the rdt3.0 sender.

**Problem 8**

The sender side of protocol rdt3.0 differs from the sender side of protocol 2.2 in that timeouts have been added. We have seen that the introduction of timeouts adds the possibility of duplicate packets into the sender-to-receiver data stream. However, the receiver in protocol rdt.2.2 can already handle duplicate packets. (Receiver-side duplicates in rdt 2.2 would arise if the receiver sent an ACK that was lost, and the sender then retransmitted the old data). Hence the receiver in protocol rdt2.2 will also work as the receiver in protocol rdt 3.0 with no change.

**Problem 9**
Problem 6

1. Suppose the sender is in state “Wait for call 1 from above” and the receiver (the receiver shown in the homework problem) is in state “Wait for 1 from below.”

2. The sender sends a packet with sequence number 1, and transitions to “Wait for ACK or NAK 1,” waiting for an ACK or NAK.

3. Suppose now the receiver receives the packet with sequence number 1 correctly, sends an ACK, and transitions to state “Wait for 0 from below,” waiting for a data packet with sequence number 0.

4. However, the ACK is corrupted. When the rdt2.1 sender gets the corrupted ACK, it resends the packet with sequence number 1.

5. Receiver receives a corrupted packet and it makes a NAK packet and sends it to the sender.

6. When the rdt2.1 sender gets the NAK, it resends the packet with sequence number 1. However, the receiver is waiting for a packet with sequence number 0 and always sends a NAK when it doesn’t get a packet with sequence number 0.

Hence the sender will always be sending a packet with sequence number 1, and the receiver will always be NAKing that packet.
Neither will progress forward from that state.

**Problem 10**

Here, we add a timer, whose value is greater than the known round-trip propagation delay. We add a timeout event to the “Wait for ACK or NAK0” and “Wait for ACK or NAK1” states. If the timeout event occurs, the most recently transmitted packet is retransmitted. Let us see why this protocol will still work with the rdt2.1 receiver.

• Suppose the timeout is caused by a lost data packet, i.e., a packet on the sender-to-receiver channel. In this case, the receiver never received the previous transmission and, from the receiver's viewpoint, if the timeout retransmission is received, it looks exactly the same as if the original transmission is being received.

• Suppose now that an ACK is lost. The receiver will eventually retransmit the packet on a timeout. But a retransmission is exactly the same action that is taken if an ACK is garbled. Thus the sender's reaction is the same with a loss, as with a garbled ACK. The rdt 2.1 receiver can already handle the case of a garbled ACK.

**Problem 13**

In a NAK only protocol, the loss of packet x is only detected by the receiver when packet N+1 is received. That is, the receiver receives N-1 and then N+1, only when N+1 is received does the receiver realize that N was missed. If there is a long delay between the transmission of N and the transmission of N+1, then it will be a long time until x can be recovered, under a NAK only protocol.

On the other hand, if data is being sent often, then recovery under a NAK-only scheme could happen quickly. Moreover, if errors are infrequent, then NAKs are only occasionally sent (when needed), and ACK are never sent — a significant reduction in feedback in the NAK-only case over the ACK-only case.

**Problem 14**

It takes 8 microseconds (or 0.008 milliseconds) to send a packet. In order for the sender to be busy 90 percent of the time, we must have

\[ util = 0.9 = (0.008n) / 30.016 \]

or \( n \) approximately 3377 packets.