There is an alternate ending to the non-consecutive repetition where the done is not required to be right after the final repetition. The sequence specification for this is shown here:

```
1  property ckDataByteCountNonConsec;
2    @(negedge ck)
3      dr |-> dr [=2:73] ##1 done;
4  endproperty
```

The timing diagram is shown in Figure 9.9 where, in this case, done is two ticks after the final dr. This still matches the sequence despite the final "##1 done" because you can think of the repetition as ending on clock tick F. There is no restriction on how long after the last tick on which dr was TRUE that done can appear. Indeed, Figure 9.8 would also match this sequence specification.

Summarizing, there are three ways to specify repetitions and ranges of repetitions of expressions:

- *Consecutive repetitions* — The expression being repeated must continuously repeat for the specified number of times. The construct "b [*8]" represents eight repetitions of b separated by ##1. The construct "b[*3:7]" represents between 3 and 7 repetitions of b separated by ##1. When a range is specified, the repetitions end when the first element of the sequence following it becomes TRUE.
- *Goto non-consecutive repetitions* — The expression being repeated can be intermittently (non-consecutively) TRUE. The construct "b[->8] ##1 done" specifies that b will be TRUE at eight possibly non-consecutive ticks and done will be TRUE strictly at the first clock tick following the last b. See Figure 9.8.
- *Non-consecutive repetitions* — This is like the goto operator except for the relationship of the last repetition in the sequence and the next part of the sequence. "b[=8] ##1 done" removes the restriction that the done must follow the last b in the next clock tick. See Figure 9.9.

The goto operator (->) is the more restrictive of the two non-consecutive types. A goto relation will only match Figure 9.8. The non-consecutive repetitions operator (=) will match both Figure 9.8 and Figure 9.9.

9.4 Calculations within Sequences

Consider the design situation where you are checking whether a sequence of data bytes is being sent correctly. For instance, you might want to insure that the sender is sending correct information on the bus. Figure 9.10 shows a protocol where this might occur.

In this protocol, data bytes are sent at consecutive clock ticks. The first data byte is on the data lines when the start signal is asserted. The last data byte, which is called the checksum is on the data lines when the done signal is asserted.

The idea behind the checksum is that it can be used to determine if the data is being received correctly. That is, the sender sends the data bytes and internally adds them up modulo 256; this sum is called the checksum. After sending the individual data bytes, the negative of the checksum byte is sent. The receiver also adds up the data bytes as they are being received. The way this works is that, in this example, the sender sends the negative