CIRCULAR BUFFERING

A digital signal processor is a specialized microprocessor for the kind of algorithms employed in digital signal processing (DSP). The main goal is to accelerate the calculations while keeping the power consumption as low as possible. In this article, we review a basic addressing capability of DSP processors, i.e. circular buffering, which allows us to significantly accelerate the data transfer in a real-time system.

Please note that since the acronym “DSP” stands for both “digital signal processing” and “digital signal processor," we will use the term “DSP processor” when referring to the hardware rather than the algorithm.

Since the finite-impulse-response (FIR) filtering is a common operation in DSP, we will continue our discussion based on examining the difference equation of an FIR filter. This simple example will show the typical properties of many DSP algorithms. After reviewing the problem of handling the incoming samples, we will discuss the circular buffering as an efficient solution to the problem.

Linear Buffering

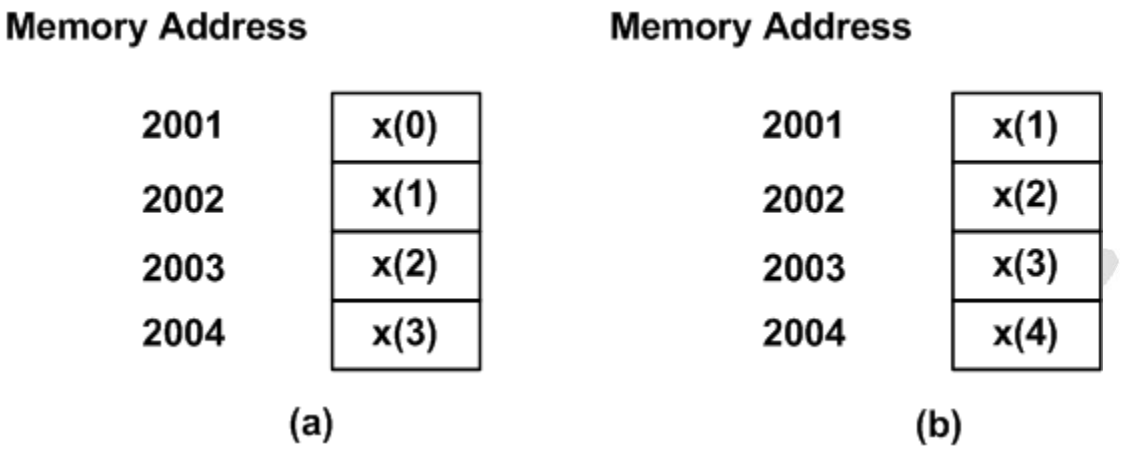

Figure (a) shows that, at time index $n = 3$ n=3, the last four samples stored in the memory are $x(3)$ x(3), $x(2)$ x(2), $x(1)$ x(1), and $x(0)$ x(0). When a new sample is acquired at $n = 4$ n=4, the last four samples, $x(4)$ x(4), $x(3)$ x(3), $x(2)$ x(2), and $x(1)$ x(1), will be stored in the memory as shown in Figure (b). This approach to storing the incoming samples is called the linear buffering. As we will see in a minute, it is simple but not at all efficient.

The main problem with linear buffering is the amount of the data transfer that we need to handle. Consider the above linear buffer for the four-tap FIR filter. With each new sample, we have to read three memory locations and write their contents to another location in our array. This is shown in Figure 4. We observe that the number of read and write operations are proportional to the length of the filter. That's why the linear buffering is not an efficient method of storing the incoming samples. For example, if we use a linear buffer to implement a 256-tap filter, then, with each new sample acquired, we have to perform almost 256 read and write operations!

Circular Buffering

Let's examine the above Figure one more time. The memory in Figure (a) stores four input samples: $x(0)$ x(0), $x(1)$ x(1), $x(2)$ x(2), and $x(3)$ x(3). Figure (b) stores $x(1)$ x(1), $x(2)$ x(2), $x(3)$ x(3) along with the new sample $x(4)$ x(4). We observe that three values, i.e. $x(3)$ x(3), $x(2)$ x(2), $x(1)$ x(1), are common between the two cases

shown in Figure 3; however different memory locations are used to store these common values.

Can we use the above observation to reduce the number of the required read and write operations? For example, what's the point of copying $x(3)$ from the hypothetical memory location 2004 in Figure (a) to the address 2003 in Figure (b), and then using it in the upcoming calculations? If we keep the common values where they currently reside, then we only need to store the new sample, i.e. $x(4)$, in the memory. When $x(4)$ is acquired, we no longer need $x(0)$, hence, we can use the memory location 2001 to store $x(4)$. The result is shown in below Figure . As shown in this figure, the newest sample, $x(4)$, replaces the oldest sample $x(0)$. This technique for storing the incoming samples is called the circular buffering. In this way, with each new sample, only a single memory write operation is required.