Signal processing with the MAXQ multiply-accumulate unit (MAC)

In the modular MAXQ architecture, a single-cycle multiply-accumulate (MAC) unit is incorporated to facilitate operation required for a typical signal-processing technique.

Traditional microcontrollers and digital signal processors (DSPs) are sometimes viewed as standing at opposite ends of the microcomputer spectrum. While microcontrollers are best suited for control applications that require low-latency response to unsynchronized events, DSPs shine in applications where intense mathematical calculations are required. A microcontroller *can* be used in heavy arithmetic applications, but the one-operation-at-a-time nature of most microcontroller ALUs makes such use less than optimal. Similarly, a DSP can be forced into a control application, but the internal architecture of most DSPs render this operation inefficient in both code and time.

Choosing a DSP or a traditional microcontroller becomes more difficult when a mostly control-oriented application requires a small amount of signal processing. In such applications, it is tempting to squeeze the DSP code into the microcontroller. However, the designer often finds that the application spends most time performing DSP functions, thus making the control application suffer.

This dichotomy can be resolved in modern processor architectures, such as the MAXQ architecture. In the modular MAXQ architecture, a multiply-accumulate unit (MAC) can be added to the design and integrated into the architecture with ease. With the hardware MAC, 16 x 16 multiply-accumulate operations occur in one cycle without compromising the application running on the control processor. This article provides some examples of how the MAC module in a typical MAXQ microcontroller can be used to solve such real-world problems.

Using the MAC module with a MAXQ

A common application for DSPs is filtering some analog signal. In this application, a properly conditioned analog signal is presented to an ADC, and the resulting stream of samples is filtered in the digital domain. A general filter implementation can be realized by the following equation:

$$y[n] = \sum b_i x[n-i] + \sum a_i y[n-i]$$

where b_i and a_i characterize the feedforward and feedback response of the system, respectively.

Depending on the values of a_i and b_i, digital filters can be classified into two broad categories: finite impulse response (FIR) and infinite impulse response (IIR). When a system does not contain any feedback elements (all $a_i = 0$), the filter is said to be of the FIR type:

$$y[n] = \sum b_i x[n-i]$$

However, when elements of both a_i and b_i are non-zero, the system is an IIR filter.

As can be seen from the above equation for an FIR filter, the main mathematical operation is to multiply each input sample by a constant, and then accumulate each of the products over the n values. The following C fragment illustrates this:

```
y[n]=0;
for(i=0; i<n; i++)
        y[n] += x[i] * b[i];
```

For a microprocessor with a multiplier unit, this can be achieved according to the following pseudo-assembler code:

```
move ptr0, #x      ;Primary data pointer -> samples
move ptr1, #b      ;Secondary DP -> coefficients
move ctr, #n       ;Loop counter gets number of samples
move result, #0    ;Clear result register
```

```
ACC_LOOP:
        move acc, @ptr0    ;Get a sample
        mul   @ptr1        ;Multiply by coefficient
        add   result       ;Add to previous result
        move  result, acc  ;...and save the result back
        inc   ptr0         ;Point to next sample
        inc   ptr1         ;Point to next coefficient
        dec   ctr          ;Decrement loop counter
        jump  nz, ACC_LOOP ;Jump if there are more samples
        end
```

The dual-tone multi-frequency (DTMF) signaling technique used in the telephone network conveys address information from a network terminal (telephone or other device) to a switch.

Thus, even with a multiplier, the multiply and accumulate loop requires 12 instructions and (assuming a one-cycle execution unit and multiplier) 4 + 8n cycles.

The MAXQ multiplier is a true multiply-accumulate unit. Performing the same operation in the MAXQ architecture shrinks code space from 12 words to 9 words, and execution time is reduced to 4 + 5n cycles.

```
        move DP[0], #x        ; DP[0] -> x[0]
        move DP[1], #b        ; DP[1] -> b[0]
        move LC[0], #loop_cnt ; LC[0] -> number of samples
        move  MCNT, #INIT_MAC ; Initialize MAC unit
MAC_LOOP:
        move DP[0], DP[0]     ; Activate DP[0]
        move MA, @DP[0]++     ; Get sample into MAC
        move DP[1], DP[1]     ; Activate DP[1]
        move MB, @DP[1]++     ; Get coeff into MAC and multiply
        djnz  LC[0], MAC_LOOP
```

Note that in the MAXQ multiply-accumulate unit, the requested operation occurs automatically when the second operand is loaded into the unit. The result is stored in the MC register. Note also that the MC register is 40 bits long, and thus can accumulate a large number of 32-bit multiply results before overflow. This improves on the traditional approach where overflow must be tested after every atomic operation. To illustrate how the MAC can be used efficiently in the signal-processing flow, we present a simple application for a dual-tone multi-frequency (DTMF) transceiver.

DTMF overview

...in the MAXQ multiply-accumulate unit, the requested operation occurs automatically when the second operand is loaded into the unit.

DTMF is a signaling technique used in the telephone network to convey address information from a network terminal (a telephone or other device) to a switch. The mechanism uses two sets of four discrete tones that are not harmonically related, i.e., the “low group” (less than 1kHz) and the “high group” (greater than 1kHz). Each digit on the telephone keypad is represented by exactly one tone from the low group and one tone from the high group. See **Figure 1** to learn how the tones are allocated.

DTMF tone encoder

The encoder portion of the DTMF transceiver is relatively straightforward. Two digital sine-wave oscillators are required, each of which can be tuned to one of the four low-group or high-group frequencies.

There are several ways to resolve the issue of digitally synthesizing a sine wave. One method of sine-wave generation avoids the issue of digital synthesis altogether. Instead, it just strongly filters a square wave produced on a port pin. While this method works in many applications, Bellcore requirements dictate that the spectral purity of the sine waves be higher than can be achieved using this technique.

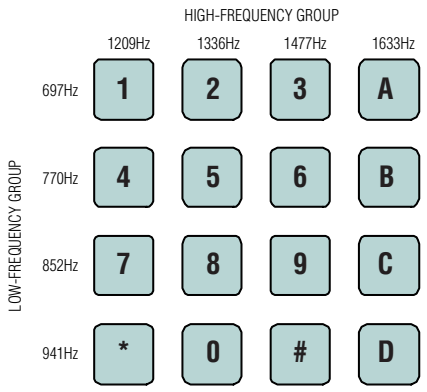

Figure 1. *Combining one frequency from the high-frequency group and one from the low-frequency group generates a DTMF signal.*

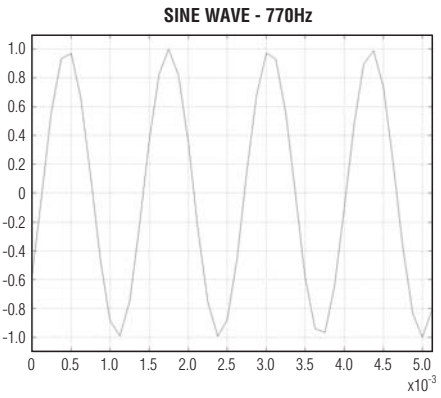

Figure 2. *A recursive resonator generates the sine wave.*

A second method of generating sinusoidal waveforms is the table-lookup method. In this method, one-quarter of a sine wave is stored in a ROM table, and the table is sampled at a precomputed interval to create the desired waveform. Creating a quarter-sine table of sufficiently high resolution to meet spectral requirements would, however, require a significant amount of storage. Fortunately, there is a better way.

A recursive digital resonator[1] can be used to generate the sinusoids (**Figure 2**). The resonator is implemented as a two-pole filter described by the following difference equation:

$$X_n = k * X_{n-1} - X_{n-2}$$

where *k* is a constant defined as

$$k = 2 \cos(2\pi * toneFrequency / samplingRate)$$

Because only a small number of tones are needed in a DTMF dialer, the eight values of k can be precomputed and stored in ROM. For example, the constant required to produce a Column 1 tone (770Hz) at a sample rate of 8kHz is:

$$k = 2 \cos(2\pi * 770 / 8000) = 2 \cos(0.60) = 1.65$$

One more value must be calculated: the initial impulse required to make the oscillator begin running. Clearly, if X_{n-1} and X_{n-2} are both zero, every succeeding X_n will be zero. To start the oscillator, set X_{n-1} to zero and set X_{n-2} to

$$X_{n-2} = -A * \sin(2\pi * toneFrequency / samplingRate)$$

In our example, assuming a unit sine wave is desired, this reduces to:

$$X_{n-2} = -1 * \sin(2\pi * 770 / 8000) = -\sin(0.60) = -0.57$$

Reducing this to code is simple: first, two intermediate variables (X1, X2) are initialized. X1 is initialized to zero, while X2 is loaded with the initial excitation value (calculated above) to start the oscillation. To generate one sample of the sinusoid, perform the following operation:

$$X0 = k * X1 - X2$$
$$X2 = X1$$
$$X1 = X0$$

The MAXQ microcontroller, together with its MAC unit, is bridging the gap between the traditional microcontroller and the digital signal processor.

Each new sine value is calculated using one multiplication and one subtraction. With a single-cycle hardware MAC on the MAXQ microcontroller, the sine wave can be generated as follows:

```
move DP[0], #X1             ; DP[0] -> X1
move MCNT, #INIT_MAC        ; Initialize MAC unit
move MA, #k                 ; MA = k
move MB, @DP[0]++           ; MB = X1, MC=k*X1, point to X2
move MA, #-1                ; MA = -1
move MB, @DP[0]--           ; MB = X2, MC=k*X1-X2, point to X1
nop                         ; wait for result
move @--DP[0], MC           ; Store result at X0
```

DTMF tone detection

Because only a small number of frequencies are to be detected, the modified Goertzel algorithm[2] is used. This algorithm is more efficient than the general DFT mechanisms and provides reliable detection of inband signals. It can be implemented as a simple second-order filter following the format in **Figure 3**.

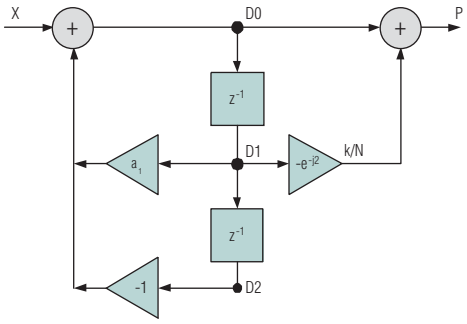

Figure 3. *The Goertzel algorithm is implemented as a second-order filter.*

To use the Goertzel algorithm to detect a tone of a particular frequency, a constant must first be precomputed. For a DTMF detector, this can be done at compile time. All the tone frequencies are well specified. The constant is computed from the following formula:

$$k = \text{toneFrequency} / \text{samplingRate}$$
$$a_1 = 2\cos(2\pi k)$$

First, three intermediate variables (D0, D1, and D2) are initialized to zero. Now, for each sample X received, perform the following:

$$D0 = X + a_1 * D1 - D2$$
$$D2 = D1$$
$$D1 = D0$$

After a sufficient number of samples has been received (usually 205 if the sample rate is 8kHz), compute the following using the latest computed values of D1 and D2:

$$P = D1^2 + D2^2 - a_1 * D1 * D2$$

P now contains a measure of the squared power of the test frequency in the input signal. To decode full four-column DTMF, each sample will be processed by eight filters. Each filter will have its own k value, and its own set of intermediate variables. Since each variable is 16 bits, the entire algorithm will require 48 bytes of intermediate storage.

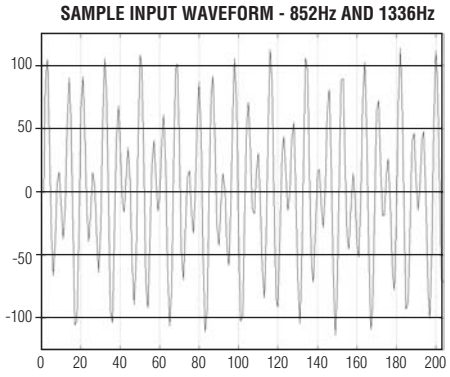

Figure 4. *This is the sample input waveform for the DTMF decoder.*

Once the P values for various tone frequencies are calculated, one tone in the high and low groups will have values significantly higher than all the other tones, which means more than twice as high, often more than an order of magnitude. **Figure 4** shows a sample input signal to the decoder, and **Figure 5** illustrates the result of the Goertzel algorithm. If the signal spectrum does not meet this criterion, it either means that no DTMF energy is present in the signal, or that there is sufficient noise to block the signal.

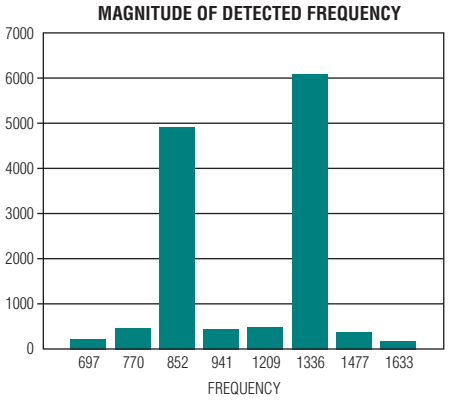

Figure 5. *The DTMF decoder detects the magnitude of various frequencies.*

A spreadsheet that demonstrates this algorithm is available on our website, as well as sample code for the MAC-equipped MAXQ processor. Go to www.maxim-ic.com/MAXQ_DTMF.

Conclusion

The MAXQ microcontroller, together with its MAC, is bridging the gap between the traditional microcontroller and the digital signal processor. With the addition of a hardware MAC, the MAXQ microcontroller offers a new level of signal-processing capability to the 16-bit microcontroller market not previously available. Real-time signal processing is made possible with a single-cycle MAC that provides the functions most often required in real-world applications.

[1] Todd Hodes, John Hauser, Adrian Freed, John Wawrzynek, and David Wessel. Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-99, March 15–19, 1999), pp. 993–996.
[2] Alan Oppenheim and Ronald Schafer, Discrete-Time Signal Processing. Prentice Hall.