Title: Digital Filtering for Atmel Mega32 (Version 2)
Author: Bruce R. Land

Introduction

Digital filtering is like auto racing: No matter how fast you can go, you would like to faster. This observation suggests that optimization of filtering algorithms for speed of execution is always in good taste. For 8-bit microcontrollers, with no floating point hardware, optimization of filtering requires tradeoffs between accuracy, ease of programming, and speed.

For ease of use and design, floating point arithmetic is the best. But when using the Atmel Mega32 microcontroller and Codevision C, a floating point multiply takes around 400 cycles and a floating add takes 120 cycles. For comparison of different techniques let's compare how many 2-pole infinite-impulse response (IIR) filters we can run in real time at an 8KHz sample rate. An 8KHz sample rate translates to 2000 cycles between samples at a maximum CPU clock rate of 16 MHz. As will be shown later, a 2-pole IIR filter requires five multiplies and four adds per sample for a total of 2480 floating point cycles. So at 8KHz sample rate we cannot even run one filter in real time using floating point arithmetic.

Going to the other extreme of simple 8-bit integer arithmetic, a one byte 8-bit by 8-bit multiply is performed in hardware in 2 cycles but takes a total of 8 cycles to fetch and store. An 8-bit add is about the same speed. We could run about 40 2-pole filters in real time, but an 8-bit format does not have the dynamic range necessary to build accurate filters. As we shall see, building accurate filters means subtracting two numbers which are almost equal, so 8-bit numbers suffer because so few bits are left after the subtraction.

A compromise is to use 16-bit fixed point arithmetic with 8-bits of integer and 8-bits of fraction (8:8 notation). As we shall see, a 16-bit fixed multiply-and-add (called a MAC operation) can be done in 24 cycles, including loading parameters. By the time registers are saved and restored in a C function call, the 2-pole IIR calculation takes 182 cycles. We can therefore perform about 11 filter operations per sample time! Accuracy is good enough for filters up to about 4-poles, but some care is needed. Narrow bandwidth may cause errors, the filter coefficients may need to be scaled, and actual filter response should always be measured. I chose 8:8 notation in spite of its accuracy limitations for three reasons: (1) It is very fast to compute. (2) We use 8:8 notation for other applications where a larger integer dynamic range is handy, specifically for video games. The integer part of the notation maps nicely into video pixels, while the 8-bit fraction is accurate enough to represent fractional pixel–change speeds. (3) The Matlab design tools that the students are used to using produce filter coefficients of up to +/-7 or so for some frequency ranges, but for smaller coefficients, it is easy to scale the Matlab-generated filter coefficients by 16 to increase fixed-point accuracy without changing the filter frequency response. Scaling required just one small change to the assembler filter code.

Students in my ECE476 microcontroller class at Cornell University (see Resources) have used these digital filters for several applications over the last two years. These include:
-- Music-controlled, stepper-motor driven Marionette.
-- MIDI synthesizer.
-- Guitar tuner/trainer.
-- Music-controlled, solenoid-driven 8-channel water fountain.
-- Voice-command car (which worked best in Hindi).
-- Voice-command lock.

Digital Filters at a high level

At a high level, digital filtering attempts to use the current input sample, past inputs and past filter outputs to select certain frequencies of the input signal. We will restrict ourselves to a subset of digital filters which are referred to as linear, time invariant filters. With this restriction, we can use powerful design tools available on the Web or in languages like Matlab to build filters (see Resources). Any linear, time invariant filter can be written as

```
a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                      - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

where y(n) is the newly computed output at time n, x(n) is the current input, x(n-1) is the previous input, y(n-1) is the previous output, with other past samples needed up to the number of b (nb) and a (na) coefficients used. The coefficient a(1) is generally set to unity in the design software. This form is referred to as "Direct Form II Transposed". The Matlab *filter* command uses this form. Determining which values for the a's and b's to use has been the object of person-centuries of research, all of which is available in any digital filtering text, from Matlab help, or from the Web. To scale the filter coefficients for better accuracy, we will set a(1) to a power of two (say 16) and multiply all the other coefficients by 16 also. This operation allows us to specify four more bits in the coefficients and only adds two cycles per bit of shift to the execution time of the filter.

For a 2-pole filter the equation reduces to

```
A(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + b(3)*x(n-2)
                  - a(2)*y(n-1) - a(3)*y(n-2)
```

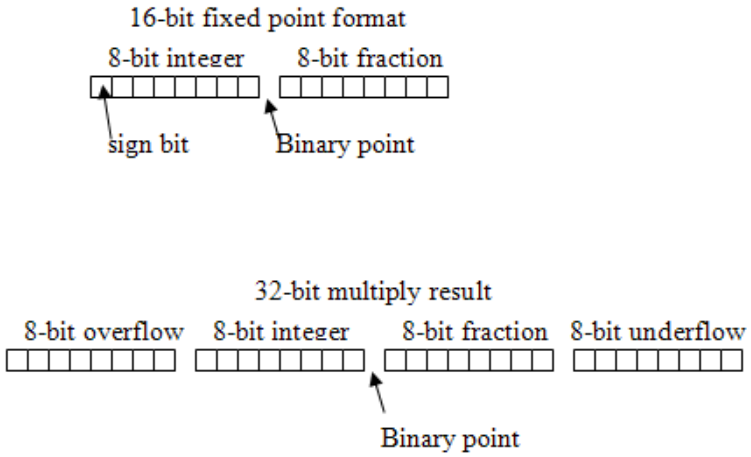
which has 5 multiplies and 4 adds as mentioned in the introduction. This formula requires that at any time, the last two past filter outputs and last two past inputs need to be available. If we used the scaled form with a(1)=16, then we will need to divide the new output value by 16, which is just four arithmetic-shift-right operations.

Fixed point Arithmetic.

Before we talk about the details of the filters, we need to talk about fixed point arithmetic. Figure 1 shows the bit layout for the number format we use. The 16-bit

number is stored in a normal, signed, 2s-complement integer variable, but the bits are interpreted (from left to right) as sign bit, 7-bits of integer (-128 to +127), and 8 bits of fraction. The left-most digit of the fraction has a weight of 0.5 while the right-most digit has a weight of .00391=1/256. We can therefore represent any number in the range of minus 128 to plus 127 to a resolution of .00391.

Figure 1



Adding two of these numbers is easy. The 2's complement add/subtract hardware simply adds them in two 8-bit pieces using an add followed by an add-with-carry instruction. Multiplication requires a little more care. Doing a 16-bit by 16-bit multiply results in a 32-bit result (Figure 1), but the output value of the fixed multiply must be another 16-bit fixed point number. The top byte of the 32-bit result is treated as an overflow and discarded. But where is the binary point in the result? Since there are two 8-bit fractions, there must be 16-bits of fraction in the output of the multiply. The lowest 8 bits is underflow. The desired 16-bit result is therefore the middle two bytes of the 32-bit result. The process can be summarized as a C language macro where x and y are 16-bit ints:

```
#define multfix(x,y) ((int)((((long)(x))*((long)(y)))>>8))
```

This macro does the multiply and is faster then a floating multiply, but is slow compared to an assembly version of the algorithm for a couple of reasons. The first is that we never have to actually compute the top byte of the 32-bit result. The second is that the 32-bit shift right can be replaced in assembler by simply copying the middle two bytes to the desired destination. Since the operation we actually want for digital filtering is a multiply followed by a running sum of five terms, we can economize further by combining the two operations into a multiply and accumulate (MAC).

The MAC operation which I wrote in assembler was based Atmel application note AVR201 entitled "Using the Hardware Multiplier". The listing below shows the details. The AVR multiply opcodes only operate on a few registers, so placement of operands is pretty much fixed with one 16-bit value in registers r22 (low byte) and r23 and the other 16-bit value in r20 (low byte) and r21. The MAC operation saves the low order result

byte in r24 so that underflow values may be added in through all five MAC operations needed for a single filter operation. The 16-bit results are in registers r30 and r31. All of the multiply opcodes put the results in r0 and r1. First a signed multiply is used to compute the most significant byte of the result which is added to r31. Then an unsigned multiply is used to compute the product of the lowest two bytes. This is followed by two signed-unsigned multiplies to compute the remaining cross terms. The MAC was written as an assembler macro to minimize execution time.

Listing 1

```
.macro mult_acc         ;r31:r30:r24 += r23:r22 * r21:r20 += p1*p2
        muls r23, r21          ; (signed)p1-high * (signed)p2-high
        add     r31, r0
        mul     r22, r20       ; p1-low * p2-low
        add     r24, r0
        adc     r30, r1
        adc     r31, r27
        mulsu   r23, r20       ; (signed) p1-high * p2-low
        add     r30, r0
        adc     r31, r1
        mulsu   r21, r22       ; (signed) p2-high * p1-low
        add     r30, r0
        adc     r31, r1
    .endm
```

With the macro written, all that had to be done was to load 5 sets of samples (input and output) and filter parameters (a's and b's) into the appropriate registers, then save the updated sample history for the next sample.

When specifying coefficients for the filters, it is convenient to have a couple of utility macros which convert floats to fixed point and back again. These macros would never be used in a fast loop, but only to define constants at the initialization or readout stages of a program where execution speed is not so important. Float2fix converts a float to a fix by shifting left 8 bits using a multiply by 256. Fix2float reverses this.

```
#define float2fix(a) ((int)((a)*256.0))
#define fix2float(a) ((float)(a)/256.0)
```

Implementing the filters

I implemented a general 2-pole IIR filter, as well as specialized 2-pole low pass, high pass and band pass Butterworth filters. I also implemented 4-pole Butterworth band pass filters. Butterworth filters have maximally flat frequency response and reasonable roll off rates. Table 1 shows execution times for the filters implemented. Symmetry in the Butterworth design allows one or two multiplies to be combined for extra efficiency. The details can be found on the fixed point math web page referenced in the Resources section. About eight 4-pole filters will run in real time at 8 KHz sample rate.

Table 1. Execution time for various filters

Filter:	Execution time

	in cycles
2-pole general IIR	182
2-pole Butterworth low/high pass	148
2-pole Butterworth band pass	140
4-pole Butterworth band pass	228

The following listing gives the source for the general 2-pole IIR filter. The Codevision C compiler allows inline assembly code which is indicated by the #asm directive. The input parameter xx is passed on the data stack, which is pointed to by the hardware registers r28 and r29. These two registers together are referred to in the assembler as the Y register. A variable declared in C may be referenced in assembler by prepending an underscore to its name. So if the variable b1 was declared in C to be of type int, then _b1 is the address at which the low byte of b1 is stored and _b1+1 is the address of the high byte. The routine pushes a couple of registers on the hardware stack, clears the MAC accumulator, performs five load-and-MAC operations, and ends by updating the sample history and restoring the two registers. The previously defined macro is inserted five times. If coefficient scaling is used, eight lines near the end should be uncommented, and the input coefficients should be multiplied by 16 before calling this routine.

Listing 2

```
int IIR2(int xx)
// xx is the current input signal sample
// returns the current filtered output sample
begin
    #asm
    push r20    ;save parameter regs
    push r21

    clr r27     ;permanent zero
    clr r24     ;clear 24 bit result reg; msb to lsb => r31:r30:r24
    clr r30
    clr r31

    lds  R22, _b1       ;load b1 from RAM
    lds  R23, _b1+1
    ld   R20, Y         ;load input parameter xx from stack
    ldd  R21, Y+1
    mult_acc            ; b1*xx

    lds  R22, _b2       ;load b2 from RAM
    lds  R23, _b2+1
    lds  R20, _xn_1     ;load x(n-1) from  RAM
    lds  R21, _xn_1+1
    mult_acc            ; b2*x(n-1)

    lds  R22, _b3       ;load b3 from RAM
    lds  R23, _b3+1
    lds  R20, _xn_2     ;load x(n-2) from  RAM
    lds  R21, _xn_2+1
    mult_acc            ; b3*x(n-2)
```

```
    lds  R22, _a2       ;load -a2 from RAM
    lds  R23, _a2+1
    lds  R20, _yn_1     ;load y(n-1) from  RAM
    lds  R21, _yn_1+1
    mult_acc            ; -a2*y(n-1)

    lds  R22, _a3       ;load -a3 from RAM
    lds  R23, _a3+1
    lds  R20, _yn_2     ;load y(n-2) from  RAM
    lds  R21, _yn_2+1
    mult_acc            ; -a3*y(n-2)

    lds  R20, _xn_1     ;load x(n-1) from  RAM
    lds  R21, _xn_1+1
    sts  _xn_2, r20      ;store x(n-2) to  RAM
    sts  _xn_2+1, R21
    ld   R20, Y         ;load input parameter xx from stack
    ldd  R21, Y+1
    sts  _xn_1, r20     ;store x(n-1) to  RAM
    sts  _xn_1+1, R21
    lds  R20, _yn_1     ;load y(n-1) from  RAM
    lds  R21, _yn_1+1
    sts  _yn_2, R20     ;store y(n-2) to  RAM
    sts  _yn_2+1, R21
;to scale the filter by 16, uncomment the next 8 lines
 ;   asr  r31  ; divide by 16 for coeff prescale
 ;   ror  r30
 ;   asr  r31
 ;   ror  r30
 ;   asr  r31
 ;   ror  r30
 ;   asr  r31
 ;   ror  r30
    sts  _yn_1, r30     ;store new output as y(n-1) to  RAM
    sts  _yn_1+1, r31

    pop r21             ;restore parameter regs
    pop r20
    #endasm
end
```

Testing the Filters

I wrote a test program to verify the frequency response of the filters. The program consists of a timer interrupt service routine (ISR) running at 8KHz and a main routine running in the background. The ISR synthesizes a sine wave at a frequency selected by the main routine. The sine wave is produced using direct digital synthesis (DDS) with a 16-bit accumulator (see Resources for more information on DDS). At every sample, either a 2-pole or 4-pole filter is executed using the sine wave as input. The ISR also measures the peak amplitude of the filter output and maintains a counter so that the main routine can measure real time. The main routine initializes the sine table necessary for DDS, the timer ISR, and the UART. It then steps through a list of test frequencies. For each frequency, main computes the DDS increment, then waits a few tenths of a second

for transients to settle, reads the maximum amplitude measured by the ISR and prints the maximum. The programs are available at a link given in the Resources section.

Because of limited accuracy of a 16-bit fixed point system (relative to the floating design tools), it is important to characterize the limitations on filter performance. Figure 2 shows the results of testing two 2-pole Butterworth band pass filters. With a bandwidth of 400 Hz (0.1 of the Nyquist frequency), the filter computed by Matlab on the PC matches the response measured on the Mega32 very closely. However with a bandwidth of 200 Hz, there is a 6% gain error (-0.5 db) because some of the coefficients are so small that very few bits are left in the terms multiplied by b1. Scaling the coefficients by 16 and using the scaled filter code reduces to error to less than 1% for the narrower bandwidth. The filter test code may be found online.

Figure 2

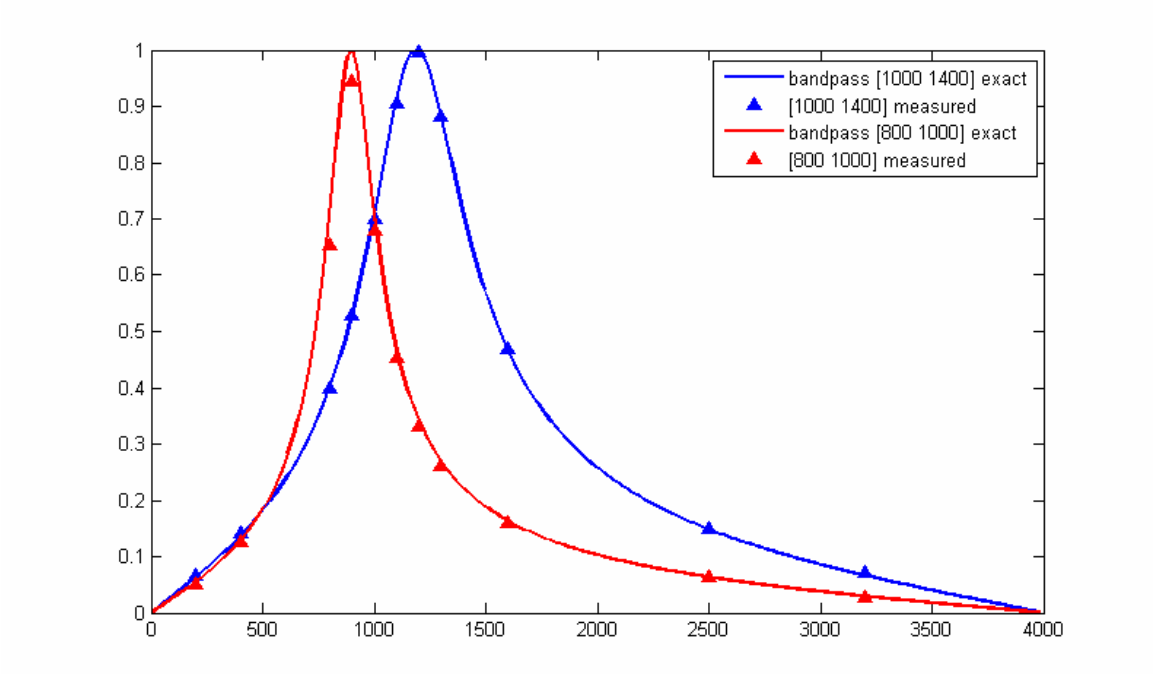


Figure 3 shows the results of testing two 4-pole Butterworth band pass filters. The filters exhibit sharper cutoffs, but more gain errors. With a bandwidth of 800 Hz (0.2 of Nyquist frequency) the filter computed by Matlab matches the response measured on the Mega32 within 2% near the peak. However with a bandwidth of 400 Hz, there is as much as an 8% gain error (0.7db) at some frequencies and 4% near the peak. Scaling the coefficients by 16 reduce the error to less than 1% for the narrower bandwidth.

Figure 3

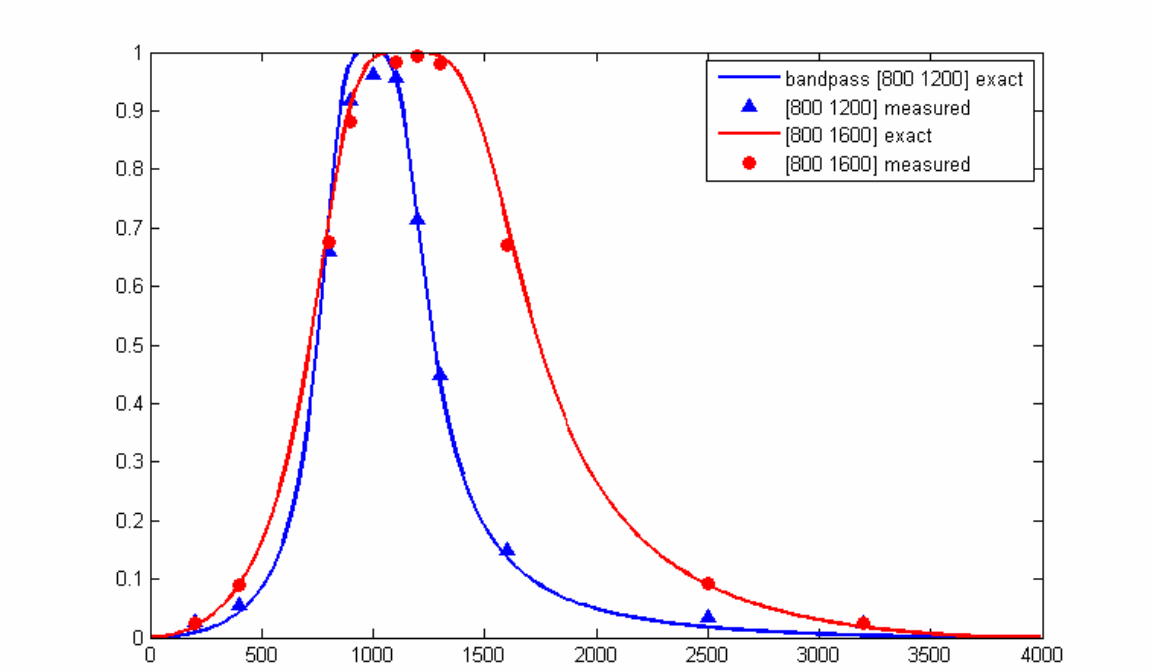
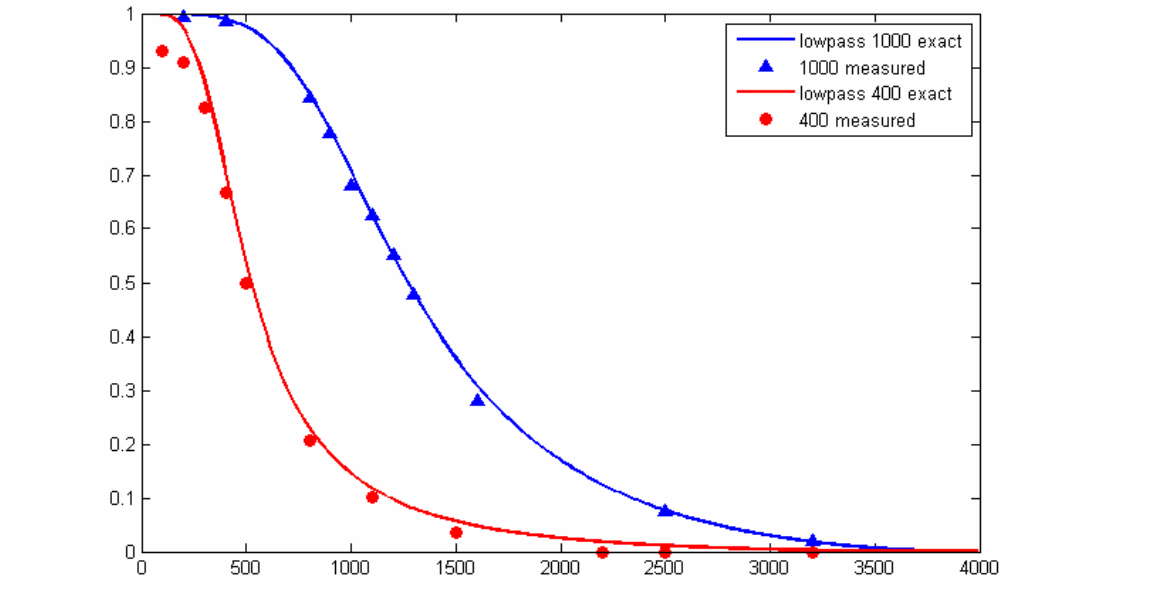


Figure 4 shows the results of testing two 2-pole Butterworth low pass filters. With a bandwidth of 1000 Hz (0.25 of Nyquist frequency) the filter computed by Matlab on the PC matches the response measured on the Mega32 very closely. At a bandwidth of 400 Hz there is a 7% gain error (-0.6 db) at low frequencies. Scaling the coefficients by 16 reduce the error to less than 1%.

Figure 4



Conclusions

The filters have been used successfully by several student projects in the course I teach (see Resources). The filters execute quickly enough to allow students to use ten general 2-pole IIR filters or eight 4-pole Butterworth band pass filters (at a sample rate of 8KHz). This is enough to do some elementary voice processing and to identify features in music.

Even the 16-bits used in fixed point format with 8 bits of fraction does not provide enough accuracy for very narrow bandwidth filters. As a rule of thumb, if you do not scale the filter coefficients, you should limit the bandwidth of 2-pole filters to greater than 200 Hz (at 8KHz sample rate) and 4-pole filters to greater than 400 Hz, in order to maintain less than 1 db gain errors. By scaling filter coefficients, you can make narrower filters. Scaling by 16 allows bandpass filters as narrow as 250 Hz at 8000 Hz sampling rate with 1 db error for 4-pole Butterworth filters. With scaled filters, dynamic range may suffer for larger valued inputs. Using only inputs with values less than one (8-bit accuracy) minimizes the chance of overflow. Narrower bandwidth filters should always be tested for adequate filter performance and for numerical overflow.

Resources.

Atmel AVR201 appnote "Using the Hardware Multiplier"
http://www.atmel.com/dyn/resources/prod_documents/DOC1631.PDF

Fixed Point Math description, including filter routines and test programs.
http://instruct1.cit.cornell.edu/courses/ee476/Math/index.html

Cornell ECE476 course and Student Projects
http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/
http://instruct1.cit.cornell.edu/courses/ee476/

Direct Digital Synthesis
Circuit Cellar #129 April 2001 page 12, Part 1: A High-Performance DDS Generator, by Robert Lacoste
and
http://www.geocities.com/CapeCanaveral/5611/dds.html

Designing Digital Filters
http://www.dsptutor.freeuk.com/IIRFilterDesign/IIRFilterDesign.html but note that the definitions of a's and b's are reversed and that the indexes are zero-based, rather than one-based. This applet assumes a sample frequency of 8000 Hz, but you can scale to other sample rates by making the *normalized cutoff frequency*,
`(cutoff frequency)/(Nyquist frequency)=(cutoff frequency)/4000`, the same.
and
http://www2.enel.ucalgary.ca/People/Turner/fpeffects/fpeffects.html Finite Precision Effects in Digital Filters, by Laurence E. Turner