Why use a DSP?

[Digital Signal Processing 101— An Introductory Course in DSP System Design—Part 2]

by David Skolnick and Noam Levine

If you've read Part 1 of this series (or are already familiar with some of the ways a DSP can work with real-world signals), you might want to learn more about how digital filters (such as those described in Part 1) can be implemented with a DSP. This article, the second of a series, introduces the following DSP topics:

- Modeling filter transform functions
- Relating the models to DSP architecture
- Experimenting with digital filters

This series seeks to describe these topics from the perspective of analog system designers who want to add DSP to their design repertoire. Using the information from articles in this series as an introduction, designers can make more informed decisions about when DSP designs might be more productive than analog circuits.

Modeling Filter Transform Functions

Part 1 compared analog and digital filter properties and suggested why one might implement these filters digitally (using DSP); this part focuses on some of the mechanics of digital filter application.

The three principal reasons for using digital filtering are (1) closer approach to ideal filter approximations, (2) ability to adjust filter characteristics in software rather than by physical tuning, and (3) compatibility of filter response with sampled data. The two best-known filters described in Part 1 are the finite impulse-response (FIR) and infinite impulse-response (IIR) types. The FIR filter response is called *finite* because its output is based solely on a finite set of input samples; it is non-recursive and has no poles, only zeroes in its *s*-plane. The IIR filter, on the other hand, has a response that can go on indefinitely (and can be unstable) because it is *recursive*, i.e., its output values are affected by both input and output. It has both poles and zeroes in its *s*-plane. Figure 1 shows the typical filter architectures and summation formulas that appeared in Part 1.

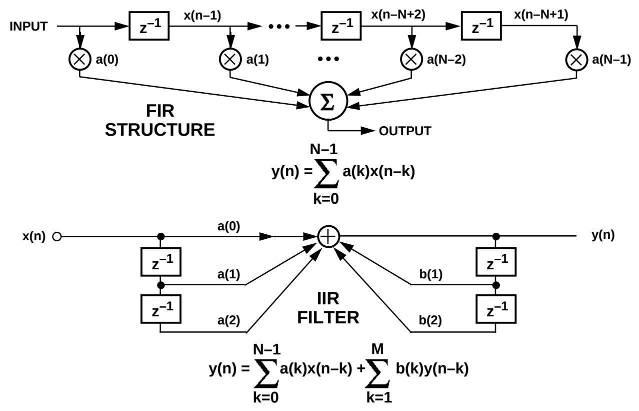


Figure 1. Filter equations and their delay-line models.

To model these filters digitally, one might take two steps. First, view these formulas as programs running on a computer. This step consists of breaking down the formula into the mathematical steps (e.g., multiply and add) and identifying all of the additional operations that would be necessary for a computer to perform (handling instructions and data, testing status, etc.) to implement the formula in software.

Second, take those operations and write them as a program. This can be a fairly arduous task. Fortunately, there is much "canned" software available, often in a high-level language (HLL) such as C, somewhat simplifying (but by no means eliminating!) the job of programming. From the point of view of learning, though, it may be more instructive to start with assembly language; also assembly language algorithms are often more useful than HLL where system performance must be optimized. At the level of abstraction of some high-level languages, the program may not look much like the equations. For example, Figure 2 shows an example of an FIR algorithm implemented as a C program.*

```
float fir_filter(float input, float *coef, int n, float *history)
{
   int i;
   float *hist_ptr, *hist1_ptr, *coef_ptr;
   float output;
   hist_ptr = history;
   hist1_ptr = hist_ptr;      /* use for history update */
   coef_ptr = coef + n -1;    /* point to last coef */
/*form output accumulation */
   output = *hist_ptr++ * (*coef_ptr-);
   for(i = 2; i < n; i++)
   {
      *hist1_ptr++ = *hist_ptr; /* update history array */
      output += (*hist_ptr++) * (*coef_ptr-);
   }
   output += input * (*coef_ptr); /* input tap */
   *hist1_ptr = input;            /* last history */
   return(output);
}
```

Figure 2. FIR Filter as C program.

There are many analysis packages available that support algorithm modeling; see the references at the end of this article for several popular packages. We will return to algorithm modeling at various times in the course of this series. Now, continuing the discussion of the process, after these filter algorithms have been modeled, they are ready for implementation in DSP architecture.

Relating The Models To DSP Architecture: For programming, one must understand four sections of DSP architecture: numeric, memory, sequencer, and I/O operations. This architectural discussion is generic (applying to general DSP concepts), but it is also specific as it relates to programming examples later in this article. Figure 3 shows the generalized DSP architecture that this section describes.

ARCHITECTURE

Numeric Section: Because DSPs must complete multiply/accumulate, add, subtract, and/or bit-shift operations in a single instruction cycle, hardware optimized for numeric operations is central to all DSP processors. It is this hardware that distinguishes DSPs from general-purpose microprocessors, which can require many cycles to complete these types of operations. In the digital filters (and other DSP algorithms), the DSP must complete multiple steps of arithmetic operations involving data values and coefficients, to produce responses in real time that have not been possible with general-purpose processors.

Numeric operations occur within a DSP's multiply/accumulator (MAC), arithmetic-logic unit (ALU), and barrel shifter (shifter). The MAC performs sum-of-products operations, which appear in most DSP algorithms (such as FIR and IIR filters and fast Fourier transforms). ALU capabilities include addition, subtraction, and

*From Embree, P. M., *C algorithms for real-time DSP*. Upper Saddle River, NJ: Prentice Hall (1995).

logical operations. Operations on bits and words occur within the shifter. Figure 3 shows the parallelism of the MAC, ALU, and shifter and how data can flow into and out of them.

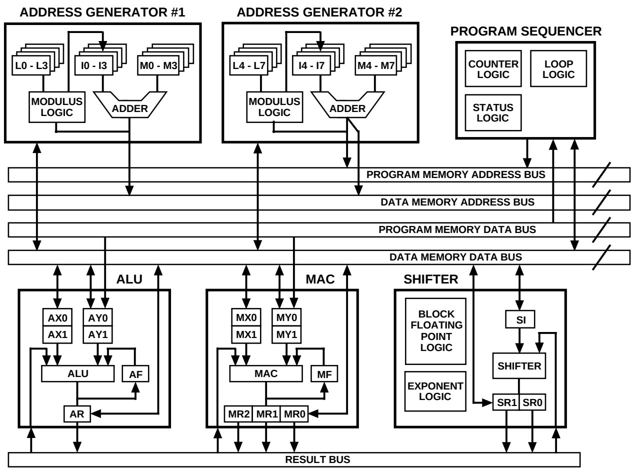


Figure 3. A useful DSP architecture.

From a programming point of view, a DSP architecture that uses separate numeric sections provides great flexibility and efficiency. There are many non-conflicting paths for data, allowing single-cycle completion of numeric operations. The architecture of the DSP must also provide a wide dynamic range for MAC operations, with the ability to handle multiplication results that are double the width of the inputs—and accumulator outputs that can mount up without overflowing. (On a 16-bit DSP, this feature equates to 16-bit data inputs and a *40-bit* result output from the MAC.) One needs this range for handling most DSP algorithms (such as filters).

Other features of the numeric section can facilitate programming in real-time systems. By making operations contingent on a variety of conditional states, which result from numeric operations, these can serve as variables in a program's execution, testing for carries, overflows, saturates, flags, or other states. Using these conditionals, a DSP can rapidly handle decisions about program flow based on numeric operations. The need to be constantly feeding data into the numeric section is a key design influence on the DSP's memory and internal bus structures.

Memory Section: DSP memory and bus architecture design is guided by the need for speed. Data and instructions must flow into the numeric and sequencing sections of the DSP on every instruction cycle. There can be no delays, no bottlenecks. Everything about the design focuses on throughput.

To put this focus on throughput in perspective, one can look at the difference between DSP memory design and memory for other microprocessors. Most microprocessors use a single memory space containing both data and instructions, using one bus for address and other for data or instructions. This architecture is called *von Neumann* architecture. The limitation on throughput in a von Neumann architecture comes from having to choose between either a piece of data or an instruction on each cycle. In DSPs, memory is typically divided into program and data memory—with separate busses for each. This type of architecture is referred to as *Harvard* architecture. By separating the data and instructions, the DSP can fetch multiple items on each cycle, doubling throughput. Additional optimizations, such as instruction cache, results feedback, and context switching also increase DSP throughput.

Etymology of Harvard and von Neumann Architectures—According to John A. N. Lee, Department of Computer Science, Virginia Tech:

> "Howard Aiken, developer of the Harvard series of machines, *insisted* on the separation of data and programs in all his machines. In the Mark III, which I know best, he even had different size drums for each."
>
> "The von Neumann concept was that by treating instructions as data one could make alterations in programs, enhancing the ability for programs to 'learn'."
>
> "For some reason, the latter was given von Neumann's name, while the former took its name from the Harvard line of machines."

Other optimizations in DSP memory architecture relate to repeated memory accesses. Most DSP algorithms, such as digital filters, need to get data from memory in a repeating pattern of accesses. Typically, this type of access serves to fetch data from a range of addresses, a range that is filled with data from the real-world signals to be processed. By reducing the number of instructions needed to "manage" memory accesses (overhead), DSPs "save" instruction cycles, allowing more time for the main job of each cycle—processing signals. To reduce overhead and automatically manage these types of accesses, DSPs utilize specialized data address-generators (DAGs).

Most DSP algorithms require two operands to be fetched from memory in a single cycle to become inputs to the arithmetic units. To supply the addresses of these two operands in a flexible manner, the DSP has two DAGs. In the DSP's modified Harvard architecture, one address generator supplies an address over the data-memory address bus; the other supplies an address over the program-memory address bus. By performing these two data fetches in time for the next numeric instruction, the DSP is able to sustain single-cycle execution of instructions.

DSP algorithms, such as the example digital filters, usually require data in a range of addresses (a buffer) to be addressed so that the address pointer "wraps-around" from the end of the buffer back to the start of the buffer (buffer *length*). This pointer movement is called *circular buffering*. (In the filter equations, each summation basically results from a sequence of multiply-and-accumulates of a circular buffer of data points and a circular buffer of coefficients). A variation of circular buffering, which is required in some applications, advances the address pointer by values greater than one address per "step," but still wraps around at a given length This variation is called *modulo circular buffering*.

By supporting various types of buffering with its DAGs, the DSP is able to perform address modify and compare operations in hardware for optimum efficiency. Performing these functions in software (as occurs in general purpose processors) limits the processor's ability to handle real-time signals.

Because buffering is an unusual concept, yet key to digital signal processing, a brief buffering example is useful. In the example illustrated in Figure 4, a buffer of eight locations resides in memory starting at address 30. The address generator must calculate next addresses that stay within this buffer yet keep the proper data spacing so that two locations are skipped. The address generator outputs the address 30 on to the address bus while it modifies the address to 33 for the next cycle's memory access. This process repeats, moving the address pointer through the buffer. A special case occurs when the address 36 gets modified to 39. The address 39 is outside the buffer. The address generator detects that the address has fallen outside of the buffer boundary and modifies the address to 31 as if the end of the buffer is connected to the start of the buffer. *The update, compare, and modify occur with no overhead.* In one cycle, the address 36 is output onto the address bus. On the next cycle, the address 31 is output onto the address bus. This modulo circular buffering serves the needs of algorithms such as interpolation filters and saves instruction cycles for processing.

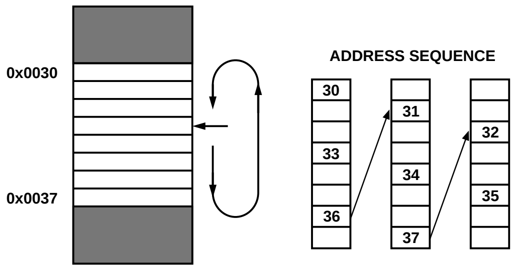
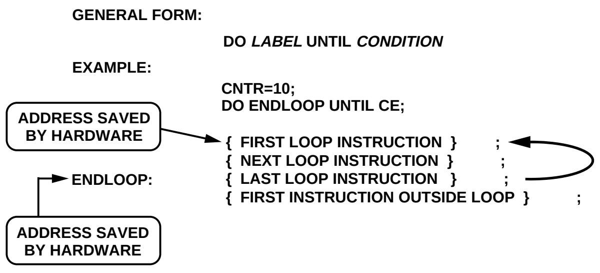


Figure 4. Example of modulo circular buffering.

Sequencer Section: Because most DSP algorithms (such as the example filters) are by nature repetitive, the DSP's program sequencer needs to loop through the repeated code without incurring overhead while getting from the end of the loop back to the start of the loop. This capability is called zero-overhead looping. Having the ability to loop without overhead is a key area in which DSPs differ from conventional microprocessors. Typically, microprocessors require that program loops be maintained in software, placing a conditional instruction at the end of the loop. This conditional instruction determines whether the address pointer moves (jumps) back to the top of the loop or to another address. Because getting these addresses from memory takes time—and availability of time for signal-processing is critical in DSP applications—DSPs cannot waste cycles retrieving addresses for conditional program sequencing (branching) in this manner. Instead, DSPs perform these test and branch functions in hardware, storing the needed addresses.

As Figure 5 shows, the DSP executes the last instruction of the loop in one cycle. On the next cycle, the DSP evaluates the conditional and executes either the first instruction at the top of the loop or the first instruction outside the loop. Because the DSP uses dedicated hardware for these operations, no extra time is wasted with software evaluating conditionals, retrieving addresses, or branching program execution.



```
DO LABEL UNTIL CONDITION

CNTR=10;
DO ENDLOOP UNTIL CE;
{ FIRST LOOP INSTRUCTION }       ;
{ NEXT LOOP INSTRUCTION }        ;
{ LAST LOOP INSTRUCTION }        ;
{ FIRST INSTRUCTION OUTSIDE LOOP }       ;
```

Figure 5. Example of program loop.

Input/Output (I/O) Section: As noted again and again, there is a need for tremendous throughput of data to the DSP; everything about its design is focused on funneling data into and out of the numeric, memory, and sequencer sections. The source of the data—and destination of the output (the result of signal processing)—is the DSP's connection to its system and the real-world. A number of I/O functions are required to complete signal processing tasks. Off-DSP memory arrays store processor instructions and data. Communication channels (such as serial ports, I/O ports and direct memory accessing (DMA) channels transfer data into and out of the DSP quickly. Other functions (such as timers and program boot logic) ease DSP system development. A brief list of typical I/O tasks in a DSP system includes the following (among many others):

- *Boot loading*: At Reset, the DSP loads instructions form an external source (EPROM or host) usually through an external memory interface.
- *Serial communications*: The DSP receives or transmits data through a synchronous serial port (SPORT), communicating with codecs, ADCs, DACs, or other devices.
- *Memory-mapped I/O*: The DSP receives or transmits data through an off-DSP memory location that is decoded by an external device.

EXPERIMENTING WITH DIGITAL FILTERS

Having modeled the filter algorithms and looked at some of the DSP architectural features, one is ready to start looking at how these filters could be coded in DSP assembly language. Up to this point the discussion and examples have been generic, applying to almost all DSPs. Here, the example is specific to the Analog Devices ADSP-2181. This processor is a fixed-point, 16-bit DSP. The term "fixed-point" means that the "point" separating the mantissa and exponent does not change its bit location during arithmetic operations. Fixed-point DSPs can be more challenging to program, but they tend to be less expensive than floating-point DSPs. The "16-bit" in "16-bit DSP" refers to the size of the DSP's data words. This DSP uses 16-bit data words and *24-bit* wide instruction words. DSPs are specified by the size of the data, rather than instruction width because data word size describes the width of data that the DSP can handle most efficiently.

The example program in Figure 6 is an FIR filter in ADSP-2181 assembly language. The software has two parts. The main routine includes register and buffer initialization along with the interrupt vector table, and the interrupt routine that executes when a data sample is ready. After initialization, the DSP executes instructions in the main routine, performing some background tasks, looping through code, or idling in a low-power standby mode until it gets an interrupt from the A/D converter. In this example, the processor idles in a low-power standby mode waiting for an interrupt.

The FIR filter interrupt subroutine (the last segment of code) is the heart of the filter program. The processor responds to the interrupt, saving the context of the main routine and jumping to the interrupt routine. This interrupt routine processes the filter input sample, reading data and filter coefficients from memory and storing them in data registers of the DSP processor. After processing the input sample, the DSP sends an output sample to the D/A converter.

```
.module/RAM/ABS=0       FIR_PROGRAM;
/******** Initialize Constants and Variables *****************/
.const                  taps=127;
.var/dm/circ            data[taps];
.var/pm/circ            fir_coefs[taps];
.init                   fir_coefs: <coeffs.dat>;
.var/dm/circ            output_data[taps];
/******** Interrupt vector table ****************************/
reset_svc:  jump start; rti; rti; rti;
                                 /*00: reset */
irq2_svc:                        /*04: IRQ2 */
            si=io(0);            /* get next sample */
            dm(i0,m0)=si;        /* store in tap delay line */
            jump fir;            /* jump to fir filter */
            nop;                 /* nop is placeholder */
irql1_svc:  rti; rti; rti; rti;  /*08: IRQL1 */
irql0_svc:  rti; rti; rti; rti;  /*0c: IRQL0 */
sp0tx_svc:  rti; rti; rti; rti;  /*10: SPORT0 tx */
sp0rx_svc:  rti; rti; rti; rti;  /*14: SPORT1 rx */
irqe_svc:   rti; rti; rti; rti;  /*18: IRQE */
bdma_svc:   rti; rti; rti; rti;  /*1c: BDMA */
sp1tx_svc:  rti; rti; rti; rti;  /*20: SPORT1 tx or IRQ1 */
sp1rx_svc:  rti; rti; rti; rti;  /*24: SPORT1 rx or IRQ0 */
timer_svc:  rti; rti; rti; rti;  /*28: timer */
pwdn_svc:   rti; rti; rti; rti;  /*2c: power down */
/******* START OF PROGRAM – initialize mask, pointers **********/
start:
      /* set up various control registers */
      ICNTL=0x07;       /* set IRQ2, IRQ1, IRQ0 edge sensitive */
      IFC=0xFF;         /* clear all pending interrupts */
      NOP;              /* add nop because of one cycle */
                        /* synchronization delay of IFC */
      SI=0x0000;
      DM(0x3FFF)=SI;    /* sports not enabled */
                        /* sport1 set for IRQ1, IRQ0, FI, FO */
      IMASK=0x200;      /* enable IRQ2 interrupt */

      i0=^data;         /* index to data buffer */
      l0=taps;          /* length of data buffer */
      m0=1;             /* post modify value */
      i4=^fir_coefs;    /* index to fir_coefs buffer */
      l4=taps;          /* length of fir_coefs buffer */
      m4=1;             /* post modify value */
      i2=^output_data;  /* index to data buffer */
      l2=taps;          /* length of data buffer */
      cntr=taps;
      do zero until ce;
        dm(i0,m0)=0; /* clear out the tap delay data buffer */
zero:   dm(i2,m0)=0; /* clear out the output_data buffer */
/**** WAIT for IRQ2 Interrupt – then JUMP to INTERRUPT VECTOR **/
wait:   idle;           /* wait for IRQ2 interrupt */
        jump wait;
/******* FIR FILTER interrupt subroutine **********************/
fir   cntr=taps-1;         /* set up loop counter */
      mr=0, mx0=dm(i0,m0), my0=pm(i4,m4);
                           /* fetch data and coefficient */
      do fir1loop until ce; /* set up loop */
fir1loop: mr=mr+mx0*my0(ss), mx0=dm(i0,m0), my0=pm(i4,m4);
                           /* calculations */
                           /*  if not ce jump fir1loop;*/
      mr=mr+mx0*my0(rnd);  /* round final result to 16-bits */
      if mv sat mr;        /* if overflow, saturate */
      io(1)=mr1;           /* send result to DAC */
      dm(i2,m0)=mr1;
      rti;
/*******  END OF PROGRAM  *************************************/
.endmod;
```

Figure 6. An FIR filter in ADSP-2181 assembly language.

Note that this program uses DSP features that perform operations with zero overhead, usually introduced by a conditional. In particular, program loops and data buffers are maintained with zero overhead. The multifunction instruction in the core of the filter loop performs a multiply/accumulate operation while the next data word and filter coefficient are fetched from memory.

The program checks the final result of the filter calculation for any overflow. If the final value has overflowed, the value is saturated to emulate the clipping of an analog signal. Finally, the context of the main routine is restored and the instruction flow is returned to the main routine with a return from interrupt (RTI) instruction.

REVIEW AND PREVIEW

The goal of this article has been to provide a link between filter theory and digital filter implementation. On the way, this article covers modeling filters with HLL programs, using DSP architecture, and experimenting with filter software. The issues introduced in this article include:

- Filters as programs
- DSP architecture (generalized)
- DSP assembly language

Because these issues involve many valuable levels of detail that one could not do justice to in this brief article, you should consider reading Richard Higgins's text, *Digital Signal Processing in VLSI*, and Paul Embree's text, *C Algorithms For Real Time DSP* (see References below). These texts provides a complete overview of DSP theory, implementation issues, and reduction to practice (with devices available at the time of publication), plus exercises and examples. The Reference section below also contains other sources that further amplify this article's issues. To prepare for the next articles in this series, you might want to get free copies of the ADSP-2100 Family User's Manual* or the ADSP-2106x SHARC User's Manual.* These texts provide information on Analog Devices's fixed- and floating-point DSP architectures, a major topic in these articles. Working through this series, each part adds some feature or information contributing to the series goal of developing a DSP system. To reach this goal, the next article describes the series' development platform (the ADSP-2181 EZ-KIT LITE) and introduces additional DSP development topics.

References

- Dearborn, G., ed., *Digital Signal Processing Applications Using the ADSP-21000 Family—Volume 1*, Norwood, MA: Analog Devices, Inc., 1994. Available from ADI. **See the book purchase card.**
- Embree, P. M., *C Algorithms for Real-Time DSP*. Upper Saddle River, NJ: Prentice Hall (1995). **Not available from ADI.**
- Higgins, R. J., *Digital Signal Processing in VLSI*, Englewood Cliffs, NJ: Prentice Hall, 1990. DSP basics. Includes a wide-ranging bibliography. Available from ADI. **See the book purchase card.**
- Mar, A., ed., *Digital Signal Processing Applications Using the ADSP-2100 Family—Volume 1*, Englewood Cliffs, NJ: Prentice Hall, 1992. Available from ADI. **See the book purchase card.**
- Mar, A., Babst, J., eds., *Digital Signal Processing Applications Using the ADSP-2100 Family—Volume 2*, Englewood Cliffs, NJ: Prentice Hall, 1994. Available from ADI. **See the book purchase card.**
- Mar, A., Rempel, H., eds., *ADSP-2100 Family User's Manual*, Norwood, MA: Analog Devices, Inc., 1995. **Free. Circle 5**
- Mar, A., Rempel, H., eds., *ADSP-21020 Family User's Manual*, Norwood, MA: Analog Devices, Inc., 1995. **Free. Circle 6**
- *MATLAB For DSP Design* (an analysis and design package for DSP), contact The Math Works, Inc. at: phone (508) 647-7000, fax: (508) 647-7101, or Web site: **http://www.mathworks.com**
- *QEDesign* (digital filter design software), contact Momentum Data Systems at: phone (714) 557-6884, fax: (714) 557-6969, or Web site: **http://www.mds.com**
- Rempel, H., ed., *ADSP-21060/62 SHARC User's Manual*, Norwood, MA: Analog Devices, Inc., 1995. **Free.**